FACULTEIT ECONOMIE EN
BEDRIJFSWETENSCHAPPEN

KATHOLIEKE
UNIVERSITEIT
LEUVEN

MODELS AND ALGORITHMS FOR SEARCH AND SEQUENCING PROBLEMS

Proefschrift Voorgedragen tot
het Behalen van de Graad van
Doctor in de Toegepaste
Economische Wetenschappen

door

Kris COOLEN

NUMMER 441 2014

Committee

Advisor:	
Prof. Dr. Roel Leus	*KU Leuven*
Co-advisor:	
Prof. Dr. Frits C. R. Spieksma	*KU Leuven*
Members:	
Prof. Dr. Dries Goossens	*Universiteit Gent*
Prof. Dr. Johann L. Hurink	*University of Twente*
Prof. Dr. Zhenbo Wang	*Tsinghua University*

Acknowledgments

Na vier-en-een-half jaar is het dan zover: er rest mij enkel nog dit dankwoord te schrijven. Het spreekwoord ‘de laatste loodjes wegen het zwaarst’ is hier wel van toepassing, lijkt me. Enerzijds omdat deze pagina’s wellicht de enigen zijn die door iedereen zal gelezen worden en daarom per definitie ook de belangrijkste zijn. Anderzijds omdat ik ongetwijfeld – en diegene die me een beetje kennen, zullen dit beamen – mensen zal vergeten vermelden. Vandaar dat ik mijn dankwoord begin met een welgemeende sorry voor alle slachtoffers van mijn verstrooidheid.

Het is de gewoonte om de eerste paragraaf te wijden aan de promotor van de doctorandus. Terecht, want aan Roel Leus heb ik enorm veel te danken. Het was Roel die me na een aantal Corona’s (of waren het Stella’s?) wist te overtuigen om voor hem te komen werken en een doctoraat voor te bereiden in ‘operationeel onderzoek’, een term die me toen nog niet bekend in de oren klonk. Het bleek al snel dat Roel een veelzijdige promotor was. Dankjewel Roel, dat je deur altijd voor me openstond en ik altijd kon binnenwandelen met mijn kleine en grote vragen. Zonder jouw enthousiasme en ideeën had deze tekst eenvoudigweg niet tot stand kunnen komen. Ook wil ik je bedanken om mij te ‘besmetten’ met je passie voor China. Ik heb genoten van onze uitstapjes naar het Verre Oosten waar je mij de Chinese thee- en eetcultuur hebt bijgebracht. Tot slot stond ik er steeds van versteld hoe je met ‘we gaan er ééntje drinken en dan gaan we slapen’ me iedere keer wist te verleiden tot menig memorabel avondje uit. Het was geweldig.

In de tweede plaats wil ik graag mijn copromotor bedanken. Ik heb Frits Spieksma leren kennen als een bescheiden en sympathiek man met een onvoorstelbare gedrevenheid en vakkundigheid. Ik heb ontzettend genoten van 'Special Topics', een vak dat deel uitmaakte van mijn doctoraal programma en dat je met veel overgave en passie doceerde. Ook was ik telkens weer onder de indruk van je vermogen om over een (moeilijk) probleem na te denken, terwijl je dit vaak pas voor het eerst hoorde, bijvoorbeeld al wandelend naar het station. Tot slot kon ik het enorm appreciëren toen je ons laatst bij je thuis had uitgenodigd voor een lekkere maaltijd voorafgegaan door een plezante quiz. De geplande wandeling viel ook reuze mee, het was minder vermoeiend dan gevreesd.

Daarnaast wil ik ook graag de leden van mijn doctoraatscommissie van harte bedanken. Bij het lezen van de tekst had Dries Goossens me er op gewezen dat mijn tabellen met rekenresultaten niet honderd procent 'koscher' waren. Dankjewel dat je hier de tijd voor hebt genomen zodat dit kon worden rechtgezet. Nu moet ik eerlijk zeggen dat Dries voor mij wel een heel erg bijzonder lid is van mijn commissie. Dries heeft gedurende mijn doctoraat namelijk op de stoel tegenover mij gezeten. Nu hij naar Gent vertrokken is, mis ik zijn ochtendlijke 'bezours', zijn aanstekelijke vertellingen over 'Club' en 'de koers', en zijn appetijt in een lekkere pizza.

Johann Hurink heb ik voor het eerst ontmoet tijdens mijn voorlopige verdediging. Ik wil hem graag bedanken omdat hij mijn doctoraatstekst erg grondig heeft gelezen. Zijn uitgebreide commentaren hebben geleid tot een beter leesbaar werk.

I would also like to express my gratitude towards Zhenbo Wang for being a member of my doctoral commission. I have enjoyed our stay at the department of mathematics of Tsinghua University in Beijing. It is highly appreciated that you have agreed to visit Leuven to be present at my public defense.

Een speciaal woord van dank gaat uit naar Martijn Huysmans. Zijn masterthesis vormt de basis voor het derde hoofdstuk van deze dissertatie.

Nu wordt het tijd om mijn collega's van het HOG te bedanken. Ik begin hierbij met de collega's van OR, en in het bijzonder met mijn andere kantoorgenoot, Fabrice Talla Nobibon. Fabrice, I have truly enjoyed you as a colleague, but also as a friend. I sincerely appreciate your scientific contributions to this work. The discussions which often took place in the cafeteria were pleasant and productive. I would also like to thank you for taking such good care of me during our foreign stays. I loved sharing a nice meal with you, and I definitely enjoyed your contagious behaviour on the dance floor. Ook dank aan mijn kantoor- en provinciegenoot, Bart, om mijn dankwoord na te lezen, met als conclusie dat ik hem vergeten was.

Next, I would like to thank Vikram Dokka en Wenchao Wei from the office next to me. They both started their PhD around the same time as I did. I enjoyed your support during the doctoral courses and also during my experience as a young researcher. Thank you Vikram for giving me the right ideas on one of the problems I have studied. I wish you all the best with your new job in Lancaster and not in the least with your wive Sonali and your little daughter. I know that it is great to be a young father, but enjoy, they grow up fast. Thank you Wenchao for your willingness to help me out, even if it meant a four-hour supervision on a Saturday morning. Good luck to you and Erin. Uiteraard mag mijn wiskunde-maatje, Daniel, hier ook niet ontbreken.

In het kantoor rechts van me zaten er twee meisjes die me nauw aan het hart liggen. Dankjewel voor de gezellige koffiepauzes. Het was geweldig om jullie passie voor zwemmen en (kleren) naaien waar te nemen. Lotte, heel veel succes met je nieuwe leven in Toulouse. Sofie, je zal binnenkort een toffe leerkracht zijn, met drie schatjes van kinderen. Houd me op de hoogste wanneer de vierde op komst is. Ondertussen is dit kantoor bemand door drie andere, maar zeker niet minder toffe collega's. Het gaat hier om het olijke duo Bart en Ward, of Ward en Bart, het is hen om het even. Ook mag ik onze nieuwe vrouwelijke aanwinst, Annette, niet vergeten.

Dan wordt het nu tijd om de collega's van 'het vierde' af te gaan. Laat

ik beginnen met de anciens, Stijn en Stefan, die toen ik begon, hun doctoraat reeds hadden afgelegd. Het was altijd weer een plezier om twee zo een uiteenlopende persoonlijkheden te zien samenwerken. Stefan, bedankt dat ik 'mijne vettige praat' bij jou kwijt kon. Dankjewel Pieter, Ann, Mieke, Yannick en Jorne dat jullie mijn 'gezever' tijdens mijn ochtendronde met onderscheiding hebben doorstaan. Verder vermeld ik graag de andere collega's van het vierde, met name Philippe, Dennis, Joeri, Valeria, Guoxuan, Stef, Raïsa, Gert, Carla en Morteza. De groep blijft groeien zodat de tafel tijdens de lunch en het aantal voorziene taarten tijdens de koffiepauze stilaan ontoereikend wordt.

Graag wil ik ook graag mijn goede vrienden bedanken. Pieter, onze nachtelijke escapades bij onze noorderburen zijn een traditie geworden die we in ere moeten houden. Pieterjan, het wordt tijd dat je met Dulce nog eens naar ons klein landje afreist. Zoniet zien wij ons genoodzaakt om jullie trouwfeest in Mexico onveilig te maken. Peter, we kennen mekaar al van het eerste middelbaar en je bent steeds een goede vriend gebleven. Onze lunchpauzes bij de Zoff waren een welkome ontspanning.

Verder mag ook zeker mijn familie hier niet ontbreken. Dankjewel mama en papa, dankzij jullie heb ik dit academisch parcours kunnen afleggen. Zonder jullie stond ik niet waar ik nu sta. Dankjewel Tine, om zo een goede zus te zijn. Ook mijn schoonouders, Jan en Gina, mogen hier niet ontbreken. Ik kon altijd bij hun terecht voor een lekkere maaltijd, een fris pintje, of een gezellige babbel.

Zoals de traditie het wil, worden de laatste woorden van dank bewaard voor de belangrijkste personen. Vera, mijn vrouwtje, dankjewel voor je eindeloze geduld en steun die je me hebt gegeven, niet in het minst wanneer het wat minder ging. Het is geweldig om elke dag naar huis te rijden, wetende dat je op mij wacht. Tot slot bedank ik mijn dochter, Hannelore, die terwijl ik deze laatste woorden schrijf, naar me lacht terwijl ik haar naam roep. Hannelore, ik houd van je, je maakt van mij de gelukkigste papa ter wereld.

Kris Coolen Leuven, februari 2014.

Table of contents

Chapter 1

Introduction

This thesis studies two different discrete optimization problems which involve sequential decisions with uncertain outcomes. The first model describes the sequential execution of a research-and-development project and is introduced in Section 1.1. This problem is called 'modular project scheduling on one machine', and is referred to as 'MP1'. We develop exact and heuristic algorithms for MP1 in Chapter 2 and Chapter 3, respectively. The second model is a specific sequential search problem that was proposed in 2001 by Wagner and Davis. It is referred to as the 'discrete sequential search problem with group activities'. We introduce this problem in Section 1.2 and briefly review a variety of related search problems that were studied in the literature. In Chapter 4, we disprove a conjecture made by Wagner and Davis on a special case of the discrete search problem with group activities (with only so-called 'conjunctive' group activities) by means of a counterexample, and further generalize this by a complexity result (NP-hardness). We conclude this introductory chapter (Section 1.3) by showing that the problems studied in Chapters 2 and 3 and those in Chapter 4 can be interpreted as two distinct, but closely related sequential search problems.

1.1 Modular project scheduling on one machine

Activities in a practical project are typically subject to many uncertainties; the most frequently studied types of uncertainty are resource breakdowns and duration variability. In research and development (R&D), activities may also fail altogether, for instance because the new technology under study does not perform as anticipated or because a toxicity test is not passed (in case of drug development). We model an R&D project as consisting of several *modules*, where a module contains one or more activities that pursue a homogeneous target, for instance representing repeated trials or technological alternatives (following Baldwin and Clark (2000)). Each activity has a cost, a duration and a probability of success. A module is successful when at least one of its included activities succeeds. The successful completion of the whole project requires the successful completion of all the modules; project success equates with receiving a project payoff (cash inflow). We subsequently refer to such projects as 'modular projects'. The objective is to schedule the activities in such a way that a maximum expected profit is attained. A solution to this scheduling problem is a *policy*, which is a dynamic decision rule that decides which activities are to be started at which time. We examine the scheduling of the project activities on a single machine, representing a scarce or bottleneck resource. Examples of such scarce resources are specialized equipment, or departments or individuals with specific areas of expertise (see Kavadias and Loch (2003) for a similar motivation in a slightly different setting). In the remainder of the thesis, we refer to this problem as MP1 (short for 'Modular Project scheduling on One machine').

1.1.1 Related work

Closely related to the model developed in Chapter 2 is the work on sequential testing, in which a series of tests is to be performed to diagnose a system (i.e., to know its state, which usually is either 'working' or 'failing').

A solution in this setting is an inspection strategy, which specifies on the basis of the state of the already inspected components which component is to be inspected next, or halts if it is able to recognize the correct state of the system. Reviews of this body of literature can be found in Boros and Ünlüyurt (1999) and Ünlüyurt (2004). The main differences with our scheduling problem are twofold: (1) the inspections will continue as long as the state of the system is not known, whereas we allow the project to be aborted preliminarily if this is better for the project's value, and (2) most of the work in this area has focused on diagnosing so-called 'k-out-of-n' systems, where the system functions if k or more of its components work. In our model, the success of a project is dependent on the success of its constituent modules, and so a project's success is not merely determined by the number of successful activities. In sequential testing, an n-out-of-n system is frequently also called a *series* system, while a 1-out-of-n system is usually referred to as a *parallel* system. The modular structure of our problem relates to a *series-parallel* system, which is a series connection of disjoint parallel subsystems. An optimal inspection strategy for a series-parallel testing system without precedence constraints is proposed by Ben-Dov (1981b). This algorithm forms the basis of the heuristic Algorithm for Problem MP1 discussed in Chapter 3.

Extensive literature surveys on the topic of scheduling under uncertainty are provided in Aytug et al. (2005); Davenport and Beck (2000); Herroelen and Leus (2005); Sabuncuoglu and Goren (2009); Vieira et al. (2003). The main topic of interest in these sources is duration uncertainty, sometimes complemented with uncertain resource availabilities. In this thesis, we incorporate the concept of activity success or failure into the scheduling decisions. De Reyck and Leus (2008) develop an algorithm for project scheduling with uncertain activity outcomes, where project success is achieved only if all individual activities succeed (series system). Ranjbar and Davari (2013) propose exact algorithms for a parallel system with arbitrary precedence constraints, but where the cash flows are discounted. Chun (1994) studies the sequencing of a set of R&D projects, rather than activities in

a single R&D project. Exact algorithms are derived for series and parallel system, but only for a very restricted type of precedence constraints among projects (they must be disjoint). A model similar to the one developed in De Reyck and Leus (2008) is tackled by Schmidt and Grossmann (1996) and Jain and Grossmann (1999), who study the scheduling of failure-prone new-product-development testing tasks when non-sequential testing is admitted. In the foregoing references, however, the possibility of pursuing multiple alternatives to achieve the same result is not included. This concept of modular projects is hinted at in the informal paper De Reyck et al. (2007), but resource constraints are not considered and no solution procedures are proposed. The work of De Reyck et al. (2007) is continued by Creemers et al. (2013), who also study modular projects but with a focus especially on the impact of activity duration variability on the project's value, whereas we work with deterministic durations. Creemers et al. (2013) also neglect resource constraints, while we are scheduling on a single machine. Malewicz (2005) studies parallel machine scheduling where tasks are executed by unreliable machines, and the probability for correct execution of each activity by each machine is known. The goal is to find a policy that assigns tasks to machines (possibly in parallel and redundantly) to minimize expected completion time; the same task can be executed more than once.

1.2 Sequential search with group activities

The *discrete sequential search problem with group activities* as defined by Wagner and Davis (2001), is as follows. A single object is hidden in one of n boxes, but the probability that a box contains that object is known for each box. The boxes are searched one at a time. When a box is searched a fixed cost is incurred (possibly different depending on the box). If the box containing the object is searched then the object is detected with certainty. When the first $n-1$ searches are negative, it is certain that the item is hidden in the final unsearched box. It is assumed that this

final box must still be searched (and therefore its cost is incurred). There are also m 'group activities'. Each group activity has also a cost and is associated with a subset of boxes. Note that some boxes may appear in more than one such subset. The group activities are said to be *conjunctive* if any box can be searched only when all the group activities in which it appears have been performed whereas for *disjunctive* group activities, a box can be searched as soon as at least one of the group activities in which it appears has been executed. The goal is to find a sequence (defining a search strategy) in which the boxes are to be searched and the group activities are to be performed so as to minimize the expected cost while satisfying the precedence constraints imposed by the group activities.

1.2.1 Related work

The problem described above is a discrete search problem with a stationary object (meaning that the object is hidden in one and only one box during the entire search process). Conventional discrete search problems with a stationary target do not include the group activities (or any other form of precedence constraints between boxes), but do incorporate the possibility of overlooking the object. Therefore it is possible that a box is searched more than once; the probability that the object is found in a box may depend on the box and on the number of times that the box was searched before. The cost of a search may also depend on the number of unsuccessful searches of that box. When there is no budget, the objective is to find a search strategy that minimizes the expected cost until the object is found (this is exactly the objective of the discrete search problem with group activities). When the budget for search is limited, the objective is typically to maximize the probability of a successful search. Another objective with a limited budget is the maximization of the probability of stating the box that holds the hidden object (whereabouts search). For more information on the results for these search problems we refer to the book by Ahlswede and Wegener (1987). A recent discrete search problem with a stationary object is studied in Song and Teneketzis (2004), where it is assumed that

there are multiple sensors available such that more than one box can be searched at each discrete time instance. The authors also point out that their problem can be viewed as a finite-horizon deterministic multi-armed bandit with multiple plays and discount factor one (Gittins, 1989). They show that under certain conditions the Gittins index rule (Gittins, 1989) coincides with their optimal search strategy. Discrete search problems with more than one stationary object have also been investigated (Assaf and Zamir, 1987). According to Stone (1989), the majority of standard search problems with a stationary object were solved by the early 1970s. That is why after this period, the focus moved to search problems with a *moving* object. Few papers assume that the movement of the target is independent of the search policy (see for example Weber (1986) or Assaf and Sharlin-Bilitzky (1994)). Other models allow intelligent targets which try to avoid being detected (Dobbie, 1975). In this setting, the problem can be seen as a game. A recent reference that uses game theory to tackle this type of problem is Owen and McCormick (2008).

1.3 Two elementary search problems

Consider the generic search model described as follows. We are given a number of boxes, say $1, \ldots, n$, each of which may have hidden within it an item. Denote by E_i the event of having an item hidden in box i and by π_i the probability of event E_i. A search of box i costs t_i. We will now briefly discuss that the two different problems studied in this thesis can be fitted into the above search model depending on the assumptions on the events E_i.

First, assume that the events are disjoint, which means that an item cannot be hidden in more than one box. Boxes are searched one by one, and the search stops when either the item is found, or all the boxes have been searched. The goal is to minimize the expected cost until the item is discovered or all boxes are searched. The assumption of disjoint events leads to a single-item discrete search problem similar to the problem described

in Chapter 4 of this thesis. Mathematically, we can express the expected cost for a given search sequence $(\sigma(1), \ldots, \sigma(n))$ as

$$\sum_{i=1}^{n} \pi_{\sigma(i)} \sum_{j=1}^{i} t_{\sigma(j)}. \tag{1.1}$$

In the second search model, we assume that the events are independent rather than disjoint. Thus it is possible that more than one box contains a hidden item, but the probability that two boxes both contain an item is equal to the product of the probabilities of each of the boxes separately. Searching ends when the first hidden item is found, or when all boxes have been searched. The expected cost for the search sequence $(\sigma(1), \ldots, \sigma(n))$ is now

$$\sum_{i=1}^{n} t_{\sigma(i)} \prod_{j=1}^{i-1} (1 - \pi_{\sigma(j)}), \tag{1.2}$$

with empty products equal to one.

Now consider the sequential testing problem for series and parallel systems as described in Section 1.1.1. We are thus given a finite number of system components, say $1, \ldots, n$. Testing component k costs c_k and determines with certainty whether the component is working or not. Denote by p_k the probability that component k is working. The goal is to determine the state (working or not working) of the overall system by a sequence of tests of its individual components against a minimum expected cost. We assume that individual components states are independent. Recall that a series system is working only if all the components are working and that a parallel system is working if at least one component is working.

It follows that testing a series or parallel system corresponds to the above search model with independent events. Indeed, for each i, we can associate a box i with a component i, and the box cost t_i with the component cost c_i. For a series system, we identify π_i with $1 - p_i$; for a parallel system, we identify π_i with p_i. By making these associations, the expected cost for the testing problem corresponds to the expected cost of the search problem given by (1.2).

MP1 with exactly one job in each module (and without precedence constraints) is equivalent to the sequential testing problem for series systems (see Theorem 2.4). This result essentially follows from the observation that in this setting the probability of project success is independent of the order in which the jobs are scheduled. Essentially the same is true for MP1 with all jobs in a single module (see Theorem 2.5). We thus conclude that MP1 can also be interpreted as a search problem.

In Kelly (1982), it is shown that the search model with disjoint events and objective (1.1) is a special case of the search model with independent events and objective (1.2) if we allow negative costs in the latter model. A similar argument is used in De Reyck and Leus (2008) to show that MP1 is NP-hard (even for series and parallel systems). The reduction is from the single machine scheduling problem with total weighted completion time and general precedence constraints. The latter problem corresponds to the search model with objective (1.1). This can be easily seen if the box costs are interpreted as the processing times, and the box probabilities as the weights (see Chapter 4 for more details). In both references, the key argument is to break up the product of probabilities into a sum. In order to establish this, probabilities are chosen arbitrarily close to one. Therefore, we need to allow large input numbers so that we can only conclude NP-hardness in the ordinary sense.

Chapter 2

Exact algorithms for MP1

In this chapter, we model a research-and-development project as consisting of several modules, with each module containing one or more activities. We examine how to schedule the activities of such a project in order to maximize the expected profit when the activities have a probability of failure and when an activity's failure can cause its module and thereby the overall project to fail. A module succeeds when at least one of its constituent activities is successfully executed. All activities are scheduled on a scarce resource that is modeled as a single machine. We describe various policy classes, establish the relations among them, develop exact algorithms to find an optimal policy in two different policy classes (one dynamic program and one branch-and-bound algorithm), and examine the computational performance of the algorithms on two randomly generated instance sets.

This chapter is the result of a collaboration with W. Wei, dr. F. Talla Nobibon and prof. dr. R. Leus. A preliminary version appeared as Research Report KBI_1124 (Coolen et al., 2011). An article containing a part of the material of this chapter is published in Journal of Scheduling (Coolen et al., 2014).

2.1 Definitions

This section starts with a number of definitions that are necessary to give a formal problem statement in the second part of the section.

2.1.1 Definitions

Consider the planning of one project in isolation, consisting of a set $N = \{0, 1, \ldots, n+1\}$ of *jobs* or *activities* (these two terms will be used interchangeably) to be scheduled on a single machine. The job set is partitioned into a set of disjoint non-empty *modules* $M = \{0, 1, \ldots, m+1\}$. Let N_i denote the set of jobs belonging to module $i \in M$, then $N = \cup_{i \in M} N_i$ and $N_i \cap N_j = \emptyset$ if $i \neq j$. Activities in the same module pursue a similar target. The (dummy) modules 0 and $m+1$ represent start and end of the project and contain only one (dummy) activity, indexed by 0 and $n+1$ respectively.

Each activity $k \in N$ has a probability of technical success (PTS) p_k such that $q_k = 1 - p_k$ is the probability of failure of that activity; we assume that $p_0 = p_{n+1} = 1$. We consider the outcomes of the different jobs to be independent. In practice, each activity also has a specific duration, but this is not relevant for the model described in this thesis because we do not consider discounting. Indeed, as pointed out in De Reyck and Leus (2008), when cash flows (costs and payoff) are not discounted, or more generally when the cash flows are time-independent, then it is a dominant decision to not schedule jobs in parallel.

A module is defined to be successful if at least one of its constituent activities succeeds. The project is said to be successful when all modules are successful.

Jobs within a module i are subjected to precedence constraints represented

by a strict partial order[1] B_i on N_i. A job in a module can only be executed when all preceding jobs of that module were scheduled before. For example, in drug development, when a certain module is needed to show the effectiveness of a drug, two precedence-related activities could represent the repeated measurement of the beneficial effects of the drug: the first test is performed after one week; the effects after two weeks will only be measured if first the effects after one week are inconclusive.

A partial order A on the set of modules M is also part of the input, and an activity in a particular module can only start when all predecessor modules are successful. Precedence constraints between modules can be both regulatory and technical in nature. Regulatory constraints often occur to protect testers or consumers: for instance, when developing a new drug the absence of toxicity has to be verified (e.g., through animal tests) before clinical tests on humans are allowed. An example of a technical precedence constraint would be the impossibility to test the toxicity of a new drug before the active ingredient has been isolated.

The foregoing definitions lead to an object $(M, A, (N_i, B_i)_{i \in M})$, which will be called the *modular network*. Furthermore, we define the order B^* on set N to relate activities that are either related in the same module or in different related modules: $(k, l) \in B^* \Leftrightarrow (\exists B_i : (k, l) \in B_i) \vee (\exists (i, j) \in A : (k \in N_i) \wedge (l \in N_j))$. The digraph with node set N and arc set B^* is referred to as the *induced network* of the modular network.

Quantity $c_k \geq 0$ represents the cost of processing activity $k \in N$; these costs are incurred at the start of each activity. We let $c_0 = c_{n+1} = 0$. The value $V > 0$ denotes the end-of-project payoff that is received at the execution of the dummy end job $n+1$; this payoff is obtained only when all mod-

[1] A strict partial order $O \subset V \times V$ defined on a set V is an asymmetric ($(i, j) \in O$ implies $(j, i) \notin O$) and transitive ($(i, j) \in O$ and $(j, l) \in O$ implies $(i, l) \in O$) relation on V. Replacing the asymmetric requirement by an irreflexive requirement ($(i, i) \notin O$) leads to an equivalent definition. Indeed, clearly the asymmetric requirement implies the irreflexive requirement. Furthermore, irreflexivity together with transitivity imply that (i, j) and (j, i) cannot belong both to O.

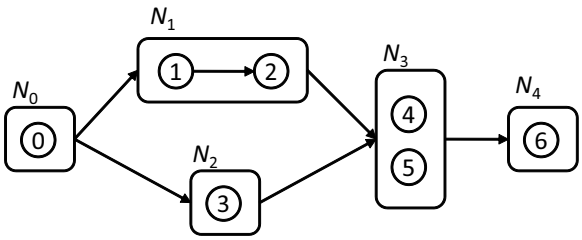
ules are successful. Our goal is to schedule the activities in order to maximize the expected profit. In the remainder of the thesis, we refer to this problem as MP1 (short for 'Modular Project scheduling on One machine'). An instance of MP1 corresponds to a tuple $(M, A, (N_i, B_i)_{i\in M}, \mathbf{p}, \mathbf{c}, V)$, with $\mathbf{p}$ and $\mathbf{c}$ two n-vectors whose components are the p_i and c_i, respectively, for $i \notin \{0, n+1\}$. Remark that we choose to drop the cost and probability information of the dummy jobs in the formal description of an MP1 instance since for any MP1 instance we assume that $p_0 = p_{n+1} = 1$ and $c_0 = c_{n+1} = 0$.

For an illustration of these definitions, we consider the instance with modular network and induced network given in Figure 2.1. The project consists of seven activities, $N = \{0, 1, 2, 3, 4, 5, 6\}$, where job 0 is the dummy start job and $n + 1 = 6$ represents the dummy end job. The jobs are partitioned into $5 = m + 2$ modules, so $M = \{0, 1, 2, 3, 4\}$ with $N_0 = \{0\}$, $N_1 = \{1, 2\}$, $N_2 = \{3\}$, $N_3 = \{4, 5\}$ and $N_4 = \{6\}$. In this example, $B_i = \emptyset$ for $i \in M\backslash\{1\}$ and $B_1 = \{(1, 2)\}$. For a binary relation R on a set S, define its *transitive closure* as the minimal transitive relation on S that contains R. The set A is the transitive closure of the set $\{(0, 1), (0, 2), (1, 3), (2, 3), (3, 4)\}$. Figure 2.1(b) shows the induced network of the modular network of Figure 2.1(a). The partial order B^* is the transitive closure of the relation

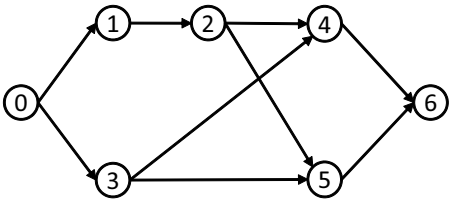
$$\{(0, 1), (0, 3), (1, 2), (2, 4), (2, 5), (3, 4), (3, 5), (4, 6), (5, 6)\}.$$

In the graphical representation of both the modular and the induced network, the transitive elements of the partial order (such as $(0, 3)$ for Figure 2.1(a) and $(1, 4)$ in Figure 2.1(b)) are not shown.

We define a *scenario* as an n-component binary vector $\mathbf{x} = (x_1, \ldots, x_n)$ with one component associated with each non-dummy activity i in N, denoting the success ($x_i = 1$) or the failure ($x_i = 0$) of activity i. Let X_i represent the Bernoulli random variable with parameter p_i of success of activity i, and denote by $\mathbf{X} = (X_1, \ldots, X_n)$ the associated vector of random variables. The realization of each X_i is known only after we perform



(a) *Modular network*



(b) *Induced network*

Figure 2.1: *Graphical representation of a modular network of an MP1 instance with five non-dummy jobs partitioned into three non-dummy modules (top) and the corresponding induced network (bottom)*

activity i (if performed at all).

A *schedule* for a project describes the sequence of activities to be executed and may vary depending on the outcomes of the scheduled jobs. A schedule may imply a *selection* of activities: not all elements of N need to be retained. This may happen, for example, when a module succeeds and remaining unexecuted jobs of that module become redundant. Therefore, a schedule $\mathbf{s}$ is an ordered subset of its non-dummy activities; by s_t we denote the job in position t in the schedule $\mathbf{s}$. Due to the precedence constraints it is clear that not all job sequences are allowed for a schedule. Firstly, jobs inside a module are only allowed to be sequenced when all preceding jobs in that module were sequenced earlier. Secondly, the precedence constraints between the modules imply that all preceding modules must succeed before the execution of the successor module is allowed. The second requirement therefore depends on the success or failure of the scheduled modules and consequently on the outcome of its individual activities.

That is why we need to specify a scenario to define the feasibility of a schedule. Formally a schedule $\mathbf{s}$ is *feasible* for scenario $\mathbf{x}$ if requirements (F1) and (F2) below hold. Requirement (F1) guarantees that precedence constraints within modules are respected, whereas requirement (F2) takes care of the precedence constraints between the modules.

(F1) for all $i \in M$, for all $l = s_u \in N_i$, and for all k with $(k, l) \in B_i$, we have $k = s_t$ for some $t < u$;

(F2) for all $i \in M$, for all $l = s_u \in N_i$, and for all j with $(j, i) \in A$, there exists $k = s_t \in N_j$ for some $t < u$ with $x_k = 1$.

Remark that, in contrast to requirement (F2), requirement (F1) does not depend on the scenario, although for our objective it is a dominant decision to schedule a job of a module only when all preceding jobs in that module have failed before (see property (R2) of a 'reasonable' policy defined in Section 2.2.1). Let $\Sigma_{\mathbf{x}}$ denote the set of all schedules feasible for $\mathbf{x}$ and let $\Sigma = \bigcup_{\mathbf{x} \in \mathbb{B}^n} \Sigma_{\mathbf{x}}$, where $\mathbb{B} = \{0, 1\}$.

A feasible schedule may not lead to a successful project for some scenarios. This may be the case when all jobs of a module have failed, or even when project success is still possible but the decision is made to abandon the project early, for example because costs or risks are too high. We will call a feasible schedule $\mathbf{s}$ *successful* for scenario $\mathbf{x}$ if a successful job is scheduled in every module (which corresponds to project success and the collection of the project payoff):

$$\forall\, i \in M \backslash \{0, m+1\}, \exists\, k = s_t \in N_i \text{ with } x_k = 1.$$

The Boolean success function $\varsigma(\mathbf{x}, \mathbf{s})$ takes value 1 if $\mathbf{s}$ is successful for $\mathbf{x}$, and 0 otherwise. For a scenario $\mathbf{x}$ and a non-empty feasible schedule $\mathbf{s}$, we define the *profit* as

$$f(\mathbf{x}, \mathbf{s}) = \varsigma(\mathbf{x}, \mathbf{s}) \cdot V - \sum_{t=1}^{|\mathbf{s}|} c_{s_t},$$

where $|\mathbf{s}|$ is the number of jobs in schedule $\mathbf{s}$. When the schedule is empty ($\mathbf{s} = \emptyset$) we set $f(\mathbf{x}, \mathbf{s}) = 0$.

For the example project described above, consider the scenario $\mathbf{x}_1 = (0, 1, 1, 0, 0)$: activities 2 and 3 succeed, but 1, 4 and 5 fail. The schedule $\mathbf{s}_1 = (1, 2, 3, 4, 5)$ is feasible for $\mathbf{x}_1$, but $\varsigma(\mathbf{x}_1, \mathbf{s}_1) = 0$: the project fails. For this scenario, there is no feasible schedule that can obtain the project payoff. The scenario $\mathbf{x}_2 = (1, 0, 1, 1, 0)$, on the other hand, allows for the payoff to be achieved: schedule $\mathbf{s}_2 = (1, 3, 4)$, for instance, is successful in this case. Note that only part of the activities in N are executed by $\mathbf{s}_2$ and that this schedule would be successful under all scenarios of the format $(1, -, 1, 1, -)$, where $-$ is either 0 or 1. When all $c_i = 1$, $i = 1, \ldots, 5$, and $V = 4$, we have $f(\mathbf{x}_1, \mathbf{s}_1) = -5$ (a negative profit of -5, or loss of 5), whereas $f(\mathbf{x}_2, \mathbf{s}_2) = 1$.

2.1.2 Problem statement

Remember from the previous section that the outcome of an activity is not known in advance, but only after it is completed. Therefore, a solution to problem MP1 is not simply one schedule, but rather a scheduling *policy*: a decision rule that decides for every possible activity outcome in which sequence to start which activities. Following Radermacher (1981) and Möhring (2000), we define a policy Π as a function $\Pi : \mathbb{B}^n \to \Sigma$, mapping scenarios $\mathbf{x}$ to feasible schedules and satisfying the *non-anticipativity constraint* (NA), which ensures that the decision made at any time t can only be based on information that became available before or at time t (in our case, time can be treated as the position of the jobs). Specifically,

(NA) if $[\Pi(\mathbf{x})]_u = l$ for an arbitrary job l and position u then also $[\Pi(\mathbf{y})]_u = l$ for all scenarios $\mathbf{y}$ that have $y_k = x_k$ for all jobs $k = [\Pi(\mathbf{x})]_t$, $t < u$.

In the foregoing, we use the notation $[\mathbf{z}]_t$ for the t^{th} component of a vector $\mathbf{z}$. In particular we have $[\Pi(\mathbf{x})]_1 = [\Pi(\mathbf{y})]_1$, $\forall\, \mathbf{x}, \mathbf{y} \in \mathbb{B}^n$. We refer to

Radermacher (1981); Stork (2001) and references therein, for more details on the use of policies as functions in stochastic scheduling.

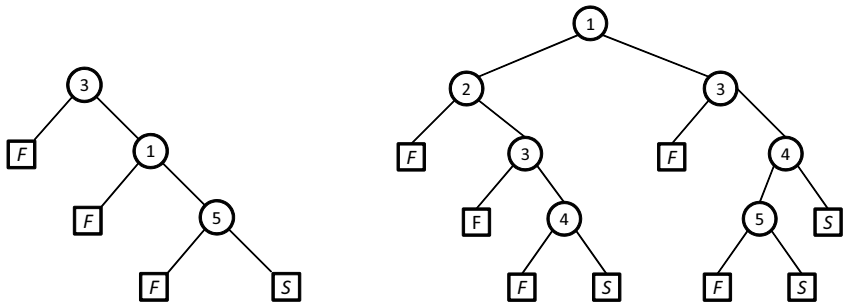
The dynamic character of a policy as a *dynamic decision process* is somewhat concealed by its representation as a function. For this reason, it is sometimes useful to adopt an alternative representation by a *binary decision tree*, which is in line with the literature on sequential testing (see Ünlüyurt (2004), for instance). In such a tree, the non-leaf nodes represent the scheduling of a non-dummy job and are labeled with the index of the job. From a non-leaf node labeled k, two decision branches emanate. The left arc represents a scenario where job k fails ($x_k = 0$) whereas the right arc implies success of job k ($x_k = 1$). The leaf nodes represent either success or failure (abandonment) of the project. To each leaf node corresponds a unique schedule: the job in position u of this schedule is precisely the label of the u-th node encountered while traversing the unique path from the root to the leaf node. When this schedule is successful, the corresponding leaf node is labeled 'S' (for success). In the other case the project is abandoned and the node is labeled 'F' (failure). For convenience, we make a slight abuse of notation in the remainder of this chapter by using the same symbol k for a node and for its corresponding job label.

Figure 2.2 shows two policies for the example project presented earlier. Policy Π_1 schedules only one job of each module and the project is abandoned as soon as a job failure is encountered. Policy Π_2 starts with job 1 and in case of failure, job 2 is executed. Depending on the outcome of job 1, module 3 is treated differently: in case of failure for job 1, only job 4 is selected whereas if $x_1 = 1$ then job 5 is started in case of failure for job 4.

The problem MP1 under study boils down to selecting a policy Π^* within a specific class $\mathcal{C}$ of policies that maximizes the expected profit of the project:

$$\Pi^* = \arg\max_{\Pi \in \mathcal{C}} \mathbb{E}[f(\mathbf{X}, \Pi(\mathbf{X}))],$$

with $\mathbb{E}[\cdot]$ the expectation operator with respect to the random variable $\mathbf{X}$. In the remainder of this text, we write $\mathbb{E}[f(\Pi)]$ instead of $\mathbb{E}[f(\mathbf{X}, \Pi(\mathbf{X}))]$



(a) *Policy* Π_1 (b) *Policy* Π_2

Figure 2.2: *Two policies for the example project of Figure 2.1(a)*

to simplify notation. As an illustration, for policy Π_1 described in Figure 2.2(a) we have

$$\mathbb{E}[f(\Pi_1)] = Vp_3p_1p_5 - c_3 - p_3c_1 - p_3p_1c_5.$$

In general terms, using the definition of a policy as a mapping, we obtain

$$\mathbb{E}[f(\Pi)] = \sum_{\mathbf{x}\in\mathbb{B}^n} \left(\prod_{i:x_i=1} p_i\right)\left(\prod_{i:x_i=0} q_i\right) f(\mathbf{x}, \Pi(\mathbf{x})).$$

Manually, the expected profit of a policy Π is more easily computed using the tree representation, T, of the policy. For an arbitrary node k, define $C(k)$ to be the set of jobs on the path from the root of T to node k (excluded), and let $C_1(k)$ and $C_0(k)$ be the subset of $C(k)$ containing only the successful and failed jobs, respectively. The collection of all the leaf nodes (respectively non-leaf nodes) of T is denoted by $L(T)$ (respectively $NL(T)$). Therefore,

$$\mathbb{E}[f(\Pi)] = \sum_{k\in L(T)} \text{Prob}(k)\left(\overline{V}_k - \sum_{l\in C(k)} c_l\right),$$

where $\text{Prob}(k) = \left(\prod_{l\in C_1(k)} p_l\right)\cdot\left(\prod_{l\in C_0(k)} q_l\right)$ is the probability of reaching node k, and

$$\overline{V}_k = \begin{cases} V & \text{if } k \text{ is labeled } S, \\ 0 & \text{if } k \text{ is labeled } F. \end{cases}$$

Equivalently,

$$\mathbb{E}[f(\Pi)] = \sum_{k \in L(T)} \text{Prob}(k) \cdot \overline{V}_k - \sum_{k \in NL(T)} \text{Prob}(k) \cdot c_k. \tag{2.1}$$

From Equation (2.1) and by stepwise updating Prob(k) from root node to leaf nodes, we have the following observation:

Observation 1. *Evaluation of an arbitrary policy can be done in time linear in the number of nodes in the decision tree.*

Since an arbitrary policy can be defined by describing its decision tree, this time complexity is the best one can hope to obtain. Furthermore, the size of the decision tree of an arbitrary policy may be exponential in the number of jobs. Evaluation of special classes of policies is discussed in Section 2.3.

We note that problem MP1 can alternatively be described as a Markov decision process (Puterman, 1994). The system state at a given decision epoch corresponds to the current progress of the project and is completely determined by the subset of jobs that are still idle at that time, where a job is said to be idle if it has not yet been started and success is not yet achieved for its module. The allowable actions in a state at a given decision epoch are twofold: either the decision maker decides to abandon the project, leading to a zero reward, or he decides to execute an eligible job from the set of remaining jobs (one that is available according to the precedence constraints). In the latter case, the decision maker receives a negative reward equal to the cost of that job, unless it is the dummy end job, in which case a positive reward is incurred corresponding with the project payoff. Finally, the transitions from a given state and selected action at a decision epoch to the state at the next decision epoch are determined by the success and failure probabilities of the jobs. In Section 2.4.1, we propose a backward stochastic dynamic-programming algorithm with state space as described in this paragraph. The value function can be computed recursively by choosing at each state the best allowable action based on the best allowable actions computed at earlier states.

2.2 Special classes of policies

Both the representation of a policy as a mapping and as a decision tree allow us to conclude that the number of scheduling policies for an MP1 instance is finite. The class of all policies is denoted by $\mathcal{C}_{ALL}$. An optimal policy in $\mathcal{C}_{ALL}$ is *globally optimal*. In this section we distinguish different policy classes, study their characteristics, and examine the relations among them. To measure the quality of a policy class for a given MP1 instance, we define the *relative optimality gap* $\gamma(\mathcal{C})$ of a policy class $\mathcal{C}$ as the relative deviation from the global optimum, i.e.

$$\gamma(\mathcal{C}) = \begin{cases} \frac{\pi(\mathcal{C}_{ALL})-\pi(\mathcal{C})}{\pi(\mathcal{C}_{ALL})} & \text{if } \pi(\mathcal{C}_{ALL}) \neq 0, \\ 0 & \text{otherwise,} \end{cases}$$

where $\pi(\mathcal{C})$ denotes the expected profit of an optimal policy in $\mathcal{C}$. We assume that the 'empty policy', which corresponds to an immediate abandonment of the project, is an element of any policy class. Therefore, $\gamma(\mathcal{C}) \in [0, 1]$. A relative gap $\gamma(\mathcal{C}) = 0$ implies that an optimal policy in $\mathcal{C}$ is also globally optimal. The other extreme, $\gamma(\mathcal{C}) = 1$, occurs when all policies in $\mathcal{C}$ have a non-positive expected profit (in which case the empty policy is an optimal policy in class $\mathcal{C}$) whereas a globally optimal policy has positive expected profit.

2.2.1 Dominance results

We define the following properties:

(R1) When a job of a module is executed with a failure and all other jobs of that module were previously scheduled without success, the project is abandoned.

(R2) When a job of a module is executed successfully, no other job of that same module is scheduled after this job.

(R3) After the success of a non-dummy activity, the project is never abandoned: in case project success is not yet achieved we schedule a new non-dummy job, otherwise we schedule the dummy end job and obtain the payoff.

A policy satisfying the properties (R1)–(R3) is called a *reasonable policy*; all reasonable policies are gathered in the set $\mathcal{C}_{REA}$. The example policies in Figure 2.2 are reasonable. We have the following dominance result:

Observation 2. *There exists a reasonable globally optimal policy for MP1, i.e.,* $\gamma(\mathcal{C}_{REA}) = 0$.

Proof. We can easily verify that an arbitrary globally optimal policy must always satisfy properties (R1)–(R3). The first property (R1) holds because it corresponds to a situation where the project cannot be completed successfully anymore. The second property (R2) also holds because unscheduled jobs of a module in which success is already achieved become redundant. Finally, the third property (R3) must also hold because otherwise it is better to not execute that job in the first place. □

Next we describe a subclass of $\mathcal{C}_{REA}$ with a specific structure. To this end, we need the following definition: a pair of distinct nodes k_1 and k_2 of the decision tree representing a reasonable policy are *equivalent* if they have an equivalent 'history', i.e., if they correspond to decision moments with the same set of successfully executed modules and the same selection of activities in the remaining modules. In other words, two distinct nodes k_1 and k_2 are equivalent if they satisfy the properties (E1)–(E2) below.

(E1) $\forall i \in M : C_1(k_1) \cap N_i \neq \emptyset \Leftrightarrow C_1(k_2) \cap N_i \neq \emptyset$.

(E2) $\forall i \in M : C_1(k_1) \cap N_i = \emptyset \Rightarrow C_0(k_1) \cap N_i = C_0(k_2) \cap N_i$.

A policy $\Pi \in \mathcal{C}_{REA}$ is called a *dominant* policy if the subtrees emerging from every pair of equivalent nodes of its decision-tree representation are

identical. The set of dominant policies is denoted by $\mathcal{C}_{DOM}$. The policy in Figure 2.2(a) is dominant because it contains no equivalent pair of nodes. The policy depicted in Figure 2.2(b), on the other hand, is not an element of $\mathcal{C}_{DOM}$; this can be seen by considering the equivalent nodes labeled with job 3: the subtrees emerging from these two nodes differ in whether or not to perform job 5 after failure of job 4. The next theorem strengthens the result of Observation 2.

Theorem 2.1. *There exists a dominant policy for MP1 that is globally optimal, i.e.,* $\gamma(\mathcal{C}_{DOM}) = 0$.

Proof. Equivalent nodes correspond to an equivalent history of the system, meaning that the remainder of the system that is yet to be scheduled is the same. Consequently, an arbitrary policy can always be transformed into a dominant policy with an expected profit that is not lower than that of the original policy by choosing the best policy for the remaining system corresponding to every pair of equivalent nodes. □

Below, we proceed with the description of a number of subclasses of $\mathcal{C}_{REA}$ that have a compact combinatorial representation, enabling simpler implementation, similar to Stork's treatment of scheduling policies for stochastic resource-constrained project scheduling (Stork, 2001). Somewhat counter-intuitively, however, we will observe in Section 2.5 that the subclasses do not allow for faster search procedures in our implementations. Section 2.2.2 presents elementary policies and module-wise policies are the subject of Section 2.2.3.

2.2.2 Elementary policies

The previously defined policy classes have the disadvantage of possibly holding solutions that are very large in size (the number of nodes in the decision tree that defines an arbitrary (dominant) policy may be exponential in the number of jobs). This hampers the communication to a practical user. Moreover, simply calculating the expected profit of an arbitrary

(dominant) policy may be computationally expensive. This motivates a study of a subclass of policies which allows a more compact representation and for which policies of that class can be computed efficiently (in polynomial time). In the remainder of this section, we restrict to policies for which the order in which jobs appear in a schedule is the same for all scenarios. We refer to such policies as *elementary* policies. Below we will show that elementary policies can be determined by a simple list of jobs. Furthermore, in the next section (Section 2.3) we proof that the expected profit of elementary policies can be computed in linear time (Theorem 2.11).

In the sequel, we use both the terms 'ordering' and '(order) list' to refer to a total order on a subset of the set $\{1, \ldots, n\}$ of non-dummy jobs. We represent an ordering of k jobs ($k \leq n$) as a permutation $L = (j_1, j_1, \ldots, j_k)$, and denote by $L(t)$ the t-th element of L, so $L(t) = j_t$. The class $\mathcal{C}_E$ of *elementary* policies is inspired by priority rules for deterministic scheduling (Kolisch, 1996a,b): each $\Pi \in \mathcal{C}_E$ is characterized by an ordering L of a subset of $N \setminus \{0, n+1\}$. We do not include the dummy jobs 0 and $n+1$ because they are always the first and the last element of the list. The ordering that defines an elementary policy is not arbitrary but should take into account the precedence constraints. We call such orderings *compatible*. Formally, a list L is compatible if either $L = \emptyset$ or the following conditions (C1)–(C4) below hold.

(C1) For each non-dummy module $i \in M \setminus \{0, m+1\}$, there is a job $k \in N_i$ in list L.

(C2) If a job l of module i belongs to L then all jobs k with $(k, l) \in B_i$ appear in the list before job l.

(C3) If a job l of module i belongs to L then for each module j with $(j, i) \in A$ there is a job of module j in the list before job l.

(C4) If a job of module i belongs to L then all jobs that appear earlier in the list are in modules j for which $(i, j) \notin A$.

Algorithm 1 Schedule generation by elementary policy $\Pi(\,\cdot\,; L)$ for scenario $\mathbf{x}$

1: $\mathbf{s} = \emptyset$
2: **while** $L \neq \emptyset$ **do**
3: Remove first job k from L and append it to the end of $\mathbf{s}$; let i be such that $k \in N_i$
4: **if** $(x_k = 0) \wedge$ (no other job of N_i appears in L) **then**
5: Return $\mathbf{s}$
6: **else if** $x_k = 1$ **then**
7: Delete all other jobs of N_i from L
8: **end if**
9: **end while**
10: Return $\mathbf{s}$

Condition (C3) is redundant because it is a consequence of (C1) and (C4). That condition is added, nevertheless, because it is needed for the definition of a 'compatible partial list' later used in Section 2.4.2), which does not require condition (C1).

Given a compatible list L, an (elementary) policy can be constructed by generating a feasible schedule for all possible scenarios. For a given scenario $\mathbf{x}$, the elementary policy $\Pi(\,\cdot\,; L)$ parameterized by compatible list L generates a unique (feasible) schedule $\Pi(\mathbf{x}; L)$ by iteratively sequencing the jobs in this list from left to right as follows: at each iteration, we check the outcome of the job; in case of success, we remove the jobs of that module that appear later in the list since they become redundant; in case of failure, we must check if there are still jobs of that module that appear in the list; if no jobs of the module are left in the list, that module cannot be finished successfully anymore, in which case the project is abandoned. Algorithm 1 describes this schedule generation procedure into more detail. Policy Π_1 in Figure 2.2(a) is elementary with $L = (3, 1, 5)$, whereas policy Π_2 in Figure 2.2(b) is not elementary due to its different treatment of jobs 4 and 5 according to the outcome of job 1. It follows from the definition of

a compatible list and Algorithm 1 that elementary policies are reasonable policies. A yet stronger result is the following:

Theorem 2.2. *Elementary policies are dominant, i.e.,* $\mathcal{C}_E \subset \mathcal{C}_{DOM}$.

Proof. The idea of the proof is to show that any pair of equivalent nodes of an elementary policy corresponds to the same job k in the list defining this policy. This is done by showing that the opposite can never occur. The theorem then follows because the subtrees that emerge from a pair of equivalent nodes are completely determined by the elements in the list that appear after job k.

Consider an elementary policy $\Pi(\,\cdot\,; L)$ for an arbitrary MP1 instance $\mathcal{I}$. We know that an elementary policy is reasonable. Choose two arbitrary equivalent nodes $k_1 = L(t_1)$ and $k_2 = L(t_2)$. If $t_1 = t_2 = t$, both subtrees must be identical as they are both completely determined by the sublist of L obtained by deleting the first $t-1$ elements of L. Next, we assume that $t_1 < t_2$ and show that this situation never occurs by deriving a contradiction for every possible occurring case. Denote by i the module containing k_1. According to (R2), $C_1(k_1) \cap N_i = \emptyset$. If we assume $k_1 \in C(k_2)$, then either $k_1 \in C_1(k_2)$ or $k_1 \in C_0(k_2)$. If $k_1 \in C_1(k_2)$ then condition (E1) implies $C_1(k_1) \cap N_i \neq \emptyset$ and a contradiction is found. If $k_1 \in C_0(k_2)$, on the other hand, then condition (E2) would imply $k_1 \in C_0(k_1)$, which is impossible. Finally, if $k_1 \notin C(k_2)$ we can choose $t < t_1$ with $L(t) \in C_1(k_2) \cap N_i$. Again by (E1), $C_1(k_1) \cap N_i \neq \emptyset$, which completes the proof. □

In line with the terminology in the sequential testing literature, define an *n*:*n*-*system* ('*n*-out-of-*n*-system') or *single-activity-module project* as an instance of MP1 where each module contains exactly one activity. For *n*:*n*-systems, every job needs to be executed successfully in order to win the project payoff. A 1:*n*-*system* ('1-out-of-*n*-system') or *single-module project*, on the other hand, contains only one non-dummy module holding all non-dummy jobs and the project succeeds at the completion of the first successful job. The following result holds:

Theorem 2.3. *Every reasonable policy for an n:n-system (and for a 1:n-system) is an elementary policy; in these cases* $\mathcal{C}_E = \mathcal{C}_{DOM} = \mathcal{C}_{REA}$.

Theorem 2.3 implies that an elementary policy exists that is globally optimal for n:n-systems (and for 1:n-systems); a result that is in line with De Reyck et al. (2007) for the setting without resource constraints. A similar result does not hold for arbitrary MP1 instances:

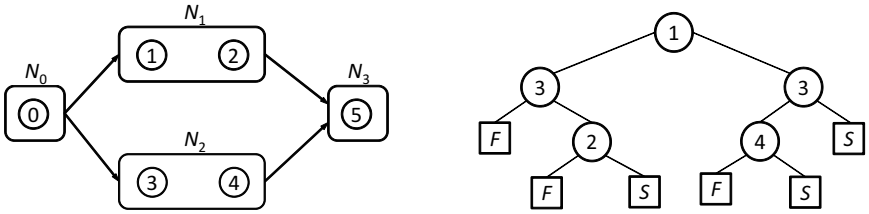
Observation 3. *There exist MP1 instances having no globally optimal elementary policy, i.e.,* $\gamma(\mathcal{C}_E) > 0$.

To verify this observation, we consider the project network in Figure 2.3(a), consisting of two parallel modules, each module containing two jobs that are not precedence-related. If we choose $p_1 = p_2 = p_3 = p_4 = \frac{1}{2}$, $c_1 = c_3 = 1$, $c_2 = c_4 = 3$ and $V = 13$, the non-elementary policy Π^* as described by Figure 2.3(b) yields a higher expected total profit than any elementary policy. A full verification of the correctness of this counterexample is presented in the appendix at the end of this chapter (page 64).

Observation 4. *There exist MP1 instances for which the best elementary policy is arbitrarily bad, i.e.,* $\gamma(\mathcal{C}_E) = 1$.

This is the case, for example, for the instance named 'g_n20_os8_10.mpo' in our data set (see Section 2.5) in which the payoff $V = 343$. For this project, the best elementary policy is the empty policy (with zero objective value), whereas a globally optimal policy with strictly positive objective value exists. Our verification of this latter result is slightly less satisfactory than for Observation 3, however, because it was assisted by a computer implementation of our algorithms (see Section 2.4) rather than by pure reasoning. We have not been able to find a counterexample of the same size as for Observation 3.

An MP1 instance with $A = \{(i,j) \,|\, i = 0 \text{ or } j = m+1\}$ and $B_i = \emptyset$ for all $i \in M$ is called *without precedence constraints*. Based on Butterworth (1972) and Mitten (1960) the special cases of n:n-systems and 1:n-systems



(a) *Graphical representation of a module network of an MP1 instance with four non-dummy jobs partitioned into two non-dummy modules*

(b) *Decision-tree representation of non-elementary policy* Π^*

Figure 2.3: *Counterexample for the claim that elementary policies would be globally optimal*

without precedence constraints are polynomially solvable. The precise results are stated in Theorem 2.4 and Theorem 2.5 below. In order to understand these results, remember that we have made the assumption that the empty policy is an element of all policy classes (see Section 2.2, page 19).

Theorem 2.4. *For an* n*:*n*-system without precedence constraints and a job list* L *with* $\frac{c_{L(k)}}{q_{L(k)}} \leq \frac{c_{L(k+1)}}{q_{L(k+1)}}$ *for all* $k = 1, \ldots, n-1$*, the elementary policy* $\Pi(\,\cdot\,; L)$ *is globally optimal unless its expected profit is less then zero, in which case it is optimal to directly abandon the project.*

Theorem 2.5. *Consider a* 1:n*-system without precedence constraints. Assume the non-dummy jobs* k *are indexed in non-decreasing ratio* c_k/p_k *and let* k^* *be the smallest job index such that* $c_{k^*}/p_{k^*} \geq V$*; if no such index exists, put* $k^* = n+1$*. The elementary policy* $\Pi(\,\cdot\,; L)$*, with* $L = (1, 2, \ldots, k^*-1)$ *if* $k^* > 1$*, and* $L = \emptyset$ *if* $k^* = 1$*, is globally optimal.*

The complexity status of MP1 without precedence constraints is still open. An adaptation of the algorithm of Ben-Dov (1981b) for series-parallel testing systems does not seem straightforward because the selection aspect leads to a circular argument: an optimal selection of jobs in each module is needed for ordering the modules, and making a selection would require

an input ordering. Furthermore, even if Ben-Dov's algorithm could be adapted, it would only produce an optimal elementary policy. The counterexample that is used to verify Observation 3, however, shows that an optimal elementary policy is not necessarily globally optimal, even for an MP1 instance without precedence constraints.

For an n:n-system with a *series-parallel* precedence graph, a polynomial algorithm exists for solving MP1 (Monma and Sidney, 1979). A series-parallel graph (SPG) is a series or parallel composition of two SPGs; furthermore, it can be verified in polynomial time whether a graph is a SPG or not (Valdes et al., 1982).

2.2.3 Module-wise policies

A *module-wise policy* or *M-policy* is a reasonable policy that only produces schedules in which all executed jobs belonging to the same module are scheduled consecutively, so no 'jumping' between modules occurs. We denote by $\mathcal{C}_M$ the class of M-policies. Formally, we have $\Pi \in \mathcal{C}_M$ if $\Pi \in \mathcal{C}_{REA}$ and for all $\mathbf{x} \in \mathbb{B}^n$, $i \in M$, and $k \in N_i$ with $x_k = 0$ and $k = [\Pi(\mathbf{x})]_t$ for some $t < |\Pi(\mathbf{x})|$, we have $[\Pi(\mathbf{x})]_{t+1} \in N_i$.

We define *module-sequence policies* or *MS-policies* as M-policies for which the order in which the modules are sequenced is the same for each possible outtcome of the activities. In other words, the module order is independent of the scenario. This class is represented by symbol $\mathcal{C}_{MS}$. More concretely, $\Pi \in \mathcal{C}_{MS}$ if $\Pi \in \mathcal{C}_M$ and there exists a strict total order $\prec$ on A such that for all $\mathbf{x} \in \mathbb{B}^n$, $i, j \in M$ with $i \neq j$, $k \in N_i$ and $l \in N_j$, with $k = [\Pi(\mathbf{x})]_s$ and $l = [\Pi(\mathbf{x})]_t$, we have $s < t \Leftrightarrow i \prec j$. Finally, the class of elementary MS-policies or EMS-policies is denoted by $\mathcal{C}_{EMS}$.

Figure 2.5(b) shows a decision-tree representation of an M-policy for MP1 instances corresponding to the network of Figure 2.5(a). This policy is not an MS-policy because depending on success or failure of activity 1, modules 2 and 3 are ordered differently. The policy of Figure 2.5(c) is

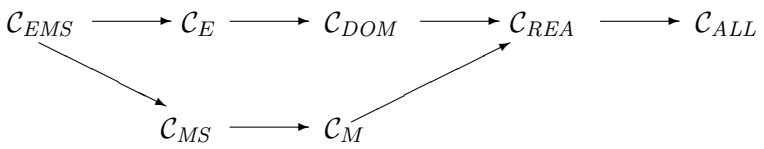


Figure 2.4: *Hierarchy of policy classes $\mathcal{C}_{ALL}$, $\mathcal{C}_{REA}$, $\mathcal{C}_{DOM}$, $\mathcal{C}_E$, $\mathcal{C}_M$, $\mathcal{C}_{MS}$ and $\mathcal{C}_{EMS}$*

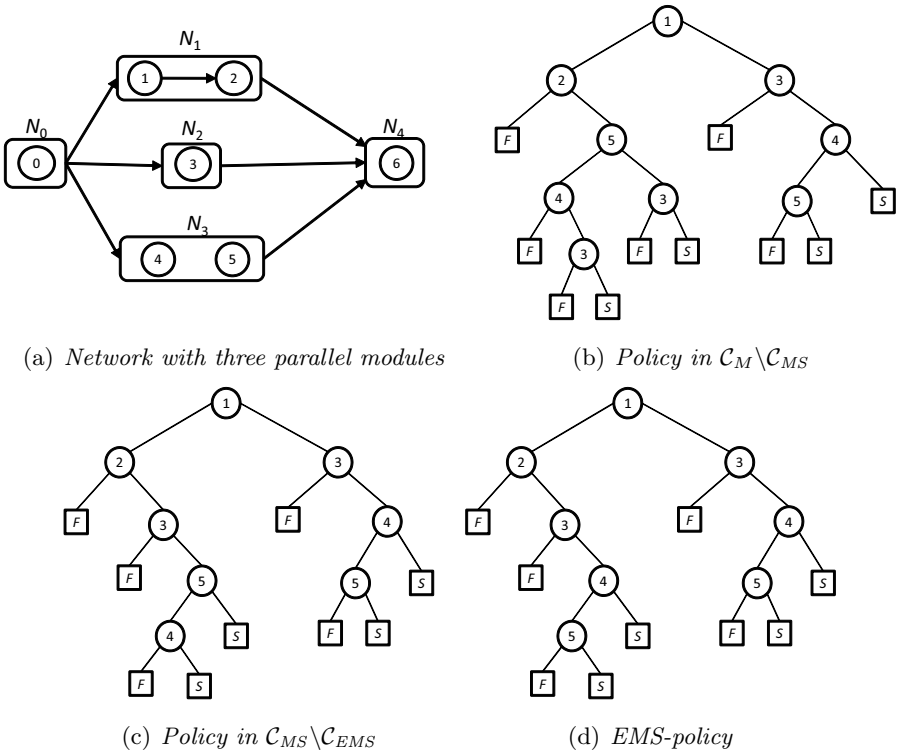
a member of C_{MS} with $1 \prec 2 \prec 3$, but it is not elementary because the execution order of jobs 4 and 5 of module 3 depends on the scenario for module 1. Figure 2.5(d) depicts an EMS-policy defined by job list $(1, 2, 3, 4, 5)$.

The hierarchy of the policy classes put forward in this section is depicted in Figure 2.4. An arrow from one class to another means that the first class is included in the second one.

In the remainder of this section, we state and prove some findings on the previously defined policy classes.

Theorem 2.6. *Dominant module-wise policies are elementary module-sequence policies and vice versa, i.e., $\mathcal{C}_{EMS} = \mathcal{C}_M \cap \mathcal{C}_{DOM}$.*

Proof. Clearly, $\mathcal{C}_{EMS} \subseteq \mathcal{C}_M \cap \mathcal{C}_{DOM}$. For a dominant M-policy Π of an arbitrary MP1 instance $\mathcal{I}$, it suffices to find a list L of the form $(L_1, \ldots, L_m)$ with $L_i \subset N_{\sigma(i)}$ for some permutation σ of the modules, such that $\Pi = \Pi(\,\cdot\,; L)$. Because Π is an M-policy, it follows that all jobs that are scheduled in the scenario where all jobs fail (jobs of the schedule $\Pi(\mathbf{0})$) must belong to the same module, say i_1. Thus, in terms of a decision tree representation T of Π, we have already established that the nodes in the upper left path of T (corresponding to job failures) correspond to jobs of a single module. Set $L_1 = \Pi(\mathbf{0})$ and for ease of notation assume $L_1 = (1, \ldots, l)$. Let $\mathcal{I}_1$ be the MP1 instance derived from $\mathcal{I}$ by deleting (all jobs of) module i_1. For a job k of L_1, consider the subtree T_k of the decision tree representation T of Π with root node equal to the node appearing at the end of the right arc (success branch) emanating from the



(a) *Network with three parallel modules*

(b) *Policy in* $\mathcal{C}_M \backslash \mathcal{C}_{MS}$

(c) *Policy in* $\mathcal{C}_{MS} \backslash \mathcal{C}_{EMS}$

(d) *EMS-policy*

Figure 2.5: *Illustration of the difference between classes* $\mathcal{C}_M$, $\mathcal{C}_{MS}$ *and* $\mathcal{C}_{EMS}$

node corresponding to job k. Denote by Π^k the policy of $\mathcal{I}_1$ defined by the subtree T_k of T. In terms of the definition of a policy as a mapping, the policy Π^k is such that $\Pi((e_k, \mathbf{x})) = ((1, \ldots, k), \Pi^k(\mathbf{x})), \forall \mathbf{x} \in \mathbb{B}^{n-l}$, with e_k a vector of $\mathbb{B}^l$ consisting of all zeros, except for component k. Since Π is dominant, all policies Π^k are identical; denote this policy by Π_1. In terms of the tree representation, all subtrees T_k are identical. Notice that Π_1 is a dominant M-policy for instance $\mathcal{I}_1$. The same procedure can be repeatedly applied to produce a list $L_j \subset N_{i_j}$, an instance $\mathcal{I}_j$, and a policy Π_j for $j = 2, \ldots, m$. Finally, set $L = (L_1, \ldots, L_m)$ and $\sigma = (i_1, \ldots, i_m)$. By construction, Π is exactly the elementary policy $\Pi(\,\cdot\,; L)$. □

Theorem 2.7. *There exists a dominant M-policy that is optimal in the class* $\mathcal{C}_M$.

Proof. Let Π be an arbitrary M-policy that is optimal in the class $\mathcal{C}_M$. Apply the procedure described in the proof of Theorem 2.1 to transform Π into a dominant policy. Recall that, in this procedure, we iteratively adapt the decision tree representation of Π by replacing the 'worse' of the two subtrees corresponding to a pair of equivalent nodes by the 'best' of these two subtrees. Hereby, the policy remains a module-wise policy in each iteration of the procedure. □

The theorem below is a direct consequence of Theorem 2.6 and Theorem 2.7.

Theorem 2.8. *There exists an EMS-policy that is optimal in the class* $\mathcal{C}_M$.

2.3 Properties

A problem of interest related to MP1 is that of searching for an optimal schedule when the outcome of the activities (failure or success) is known in advance. An instance of this problem corresponds to a tuple $(M, A, (N_i, B_i)_{i\in M}, \mathbf{x}, \mathbf{c}, V)$ in which $\mathbf{x}$ is a given scenario, contrary to an MP1 instance, where a vector $\mathbf{p}$ containing the success probabilities is given. The objective is to find a feasible schedule $\mathbf{s}^*$ that maximizes $f(\mathbf{x}, \mathbf{s}^*)$. We refer to this problem as DMP1 (short for 'Deterministic Modular Project scheduling on One machine'). We have the following result.

Theorem 2.9. *DMP1 can be solved in polynomial time.*

Proof. If there is a module in which all activities fail, $\mathbf{s}^* = \emptyset$ is optimal because in this case all non-empty policies have a non-positive expected profit. Otherwise, it is essentially optimal to select in each module the 'best' successful job. Since the outcomes are known in advance, it is clear that we only need to consider successful jobs for which all preceding jobs in that module are unsuccessful (because clearly stopping at the 'first' successful job in a module is a dominant decision). Formally, for each

non-dummy module i, define N_i' as the set of jobs $l \in N_i$ with $x_l = 1$ and $x_k = 0$ for all jobs $k \in N_i$ with $(k,l) \in B_i$. For each job $l \in N_i'$ define $\alpha_l := c_l + \sum_{k:(k,l)\in B_i} c_k$. Choose for each non-dummy module i a job, say $l^*(i)$, such that $l^*(i) = \arg\min_{l\in N_i'} \alpha_l$. Next, we choose a linear extension of the partial order A, and order the jobs $l^*(i)$ $(i = 1, \ldots, m)$ according to the chosen module order. This results is an ordered job set, say $\mathbf{s}$, containing all the $l^*(i)$ such that $(i,j) \in A$ implies $l^*(i)$ before $l^*(j)$ in $\mathbf{s}$. In order to guarantee that $\mathbf{s}$ is a feasible schedule we insert all jobs k for which $(k, l^*(i)) \in B_i$ immediately before the corresponding $l^*(i)$ in an order that respects B_i. If $f(\mathbf{x}, \mathbf{s}) > 0$ then $\mathbf{s}^* = \mathbf{s}$ is optimal, otherwise $\mathbf{s}^* = \emptyset$ is optimal. The described procedure produces an optimal schedule in polynomial time. □

For the general MP1 problem, on the other hand, the following complexity result holds:

Theorem 2.10. *MP1 is NP-hard, even for the special cases of n:n-systems and 1:n-systems.*

Proof. For n:n-systems, the proof can be found in De Reyck and Leus (2008); the authors use a reduction from the single-machine scheduling problem with precedence constraints and total weighted completion-time objective, which is NP-hard (Garey and Johnson, 1979). NP-hardness for the 1:n case follows from a reduction from n:n-systems to 1:n-systems (De Reyck et al., 2007). □

Another important issue is the hardness of computing the profit of an elementary policy for an arbitrary MP1 instance, which is settled by the following theorem:

Theorem 2.11. *Given an arbitrary MP1 instance $\mathcal{I}$ and an arbitrary job list L compatible with $\mathcal{I}$, the expected profit $\mathbb{E}[f(\Pi(\,\cdot\,; L))]$ of the elementary policy $\Pi(\,\cdot\,; L) \in \mathcal{C}_E$ can be computed in time linear in n.*

Proof. In general, the expected profit of a policy Π can be obtained as

$$\mathbb{E}([f(\Pi)]) = S \cdot V - \sum_{k=1}^{n} R(k)c_k, \tag{2.2}$$

where S is the probability of project success and $R(k)$ the probability that job k is paid for when policy Π is applied. The probabilities $R(k)$ can be calculated recursively when the policy is elementary. Obviously, $R(k) = 0$ if $k \notin L$. For job k in position t of the list L, denote with i_t the module of job $L(t)$, i.e., $k = L(t) \in N_{i_t}$, $t \in \{1, \ldots, T\}$ with $T = |L|$. In this case we have $R(k) = (1 - \pi_{i_t}(t))(1 - Q(t))$, with $Q(t)$ the probability that the project fails before the t-th job in L is processed and $\pi_{i_t}(t)$ the probability that success is achieved for module i_t before job $L(t)$ is executed, under the condition that the project is not abandoned before position t. We initialize $Q(1) = 0$ and $\pi_j(1) = 0$ for all $j = 1, \ldots, m$, because no non-dummy activities have been started yet before $L(1)$. We extend the definition of $Q(t)$ and $\pi_j(t)$ to $t = T + 1$, which corresponds to adding the dummy end job to the end of the list, i.e., $L(T + 1) = n + 1$. Because the project is successful if and only if every module is successful, we have $S = \prod_{j=1}^{m} \pi_j(T + 1) = 1 - Q(T + 1)$. For $1 \leq j \leq m$ and $2 \leq t \leq T + 1$ the following recursion for $\pi_j(t)$ and $Q(t)$ hold:

$$\pi_j(t) = \begin{cases} \pi_j(t-1) + (1 - \pi_j(t-1))\, p_{L(t-1)} & \text{if } j = i_{t-1}, \\ \pi_j(t-1) & \text{otherwise,} \end{cases} \tag{2.3}$$

$$Q(t) = \begin{cases} Q(t-1) + (1 - Q(t-1))\,(1 - \pi_{i_{t-1}}(t)) \\ \qquad \text{if } N_{i_{t-1}} \cap \cup_{s \geq t} L(s) = \emptyset, \\ Q(t-1) \quad \text{otherwise.} \end{cases} \tag{2.4}$$

Equation (2.3) states that the probability of achieving success in a module j before the job in position t in the list is executed (under the condition that the project is not abandoned before position t), only differs from the

probability of achieving success in that module before the job in position $t-1$ in the list is executed (under the condition that the project is not abandoned before position $t-1$), when the job in position $t-1$ also belongs to module j. In this case the probability $\pi_j(t)$ increases when the job in position $t-1$ is successful and the module is not yet successful before the execution of that job.

Equation (2.4) states that the probability of project abandonment before the execution of the job in position t in the list, only differs from the probability of project abandonment before the execution of the job in position $t-1$ in the list, if the job in position $t-1$ is the only job left in the list of that module. In this case the project abandonment probability $Q(t)$ increases when the project was not yet abandoned before the execution of the job in position $t-1$ and when success was not yet achieved in the module of the job in position $t-1$ before its execution.

For a given position t, we need $Q(t)$ and $\pi_j(t)$ for only one module j to know the value of $R(L(t))$. Moreover, to obtain $\pi_j(t)$ for all modules j, we only need to adapt $\pi_j(t-1)$ for one module j. Combined with the observation that a recursive step takes only constant computation time, the theorem follows. □

For general policies, by contrast, evaluation time may be exponential in the number of jobs because the number of nodes in the decision tree may be exponential in n and a compact representation is not always at hand (see Observation 1). Finally, we mention that the values $\pi_j(t)$ can alternatively be computed directly, as follows:

$$\pi_j(t) = \begin{cases} 0 & \text{if } \nexists\, L(s) \in N_j,\ s < t \\ 1 - \prod_{L(s)\in N_j,\, s<t} q_{L(s)} & \text{otherwise.} \end{cases}$$

As an example, consider job list $L = (1, 3, 2, 4)$ for the network of Figure 2.3(a). Note that the equality $p_k + q_k p_l = 1 - q_k q_l$ holds for every pair of jobs k, l. From Table 2.1 we find $R(1) = 1$, $R(3) = 1$, $R(2) = q_1$,

Table 2.1: *Computation of values Q and π*

position t	1	2	3	4	5
$L(t)$	1	3	2	4	5
$\pi_1(t)$	0	p_1	p_1	$p_1 + q_1p_2$	$p_1 + q_1p_2$
$\pi_2(t)$	0	0	p_3	p_3	$p_3 + q_3p_4$
$Q(t)$	0	0	0	q_1q_2	$q_1q_2 + (1 - q_1q_2)q_3q_4$

$R(4) = q_3(1 - q_1q_2)$, and $S = (1 - q_1q_2)(1 - q_3q_4)$, leading to

$$\mathbb{E}[f(\Pi(\,\cdot\,; L))] = (1 - q_1q_2)(1 - q_3q_4)V - c_1 - c_3 - q_1c_2 - q_3(1 - q_1q_2)c_4.$$

One easily verifies that the same result is obtained via the decision-tree representation and Equation (2.1). Alternative implementations to achieve the same time complexity as Theorem 2.11 for n:n-systems and 1:n-systems and EMS-policies are given below:

- Consider an n:n-system and let $L = (1, \ldots, n)$ be the list determining the elementary policy $\Pi(\,\cdot\,; L)$. According to Equation (2.1), the expected profit $\mathbb{E}[f(\Pi(\,\cdot\,; L))]$ equals

$$V \prod_{k=1}^{n} p_k - c_1 - \sum_{k=2}^{n} \left(\prod_{l=1}^{k-1} p_l \right) c_k.$$

  We have $O(n)$ additive operations, but the number of multiplications is $O(n^2)$. Nevertheless, we can compute the numbers $a_l := \prod_{k=1}^{l} p_k, l = 1, \ldots, n$, only using $O(n)$ multiplications because $a_l = a_{l-1}p_l$ $(l = 2, \ldots, n)$. A similar reasoning can be followed for the evaluation of elementary policies for 1:n-systems.

- Consider an arbitrary EMS-policy Π with corresponding job list L and assume a total module order $\prec$ such that $1 \prec \cdots \prec m$. For each module $j \in \{1, \ldots, m\}$, let $\mathcal{I}_j$ be the instance of MP1 consisting of only module j and define a list L_j as the sublist of L consisting of jobs

of module j only. Policy $\Pi(\,\cdot\,; L_j)$ is a well defined elementary policy for instance $\mathcal{I}_j$; remark that $\mathcal{I}_j$ is a 1:$|L_j|$-system. Now recursively define a_j to be the expected profit of $\Pi(\,\cdot\,; L_j)$ with payoff equal to a_{j+1} and initialize $a_{m+1} \equiv V$. Because of the structure of the EMS-policy Π, we have $\mathbb{E}[f(\Pi)] = a_1$ and in order to obtain a_1 we need to compute the expected profit of the elementary policies $\Pi(\,\cdot\,; L_j)$ for $j = m, m-1, \ldots, 1$. Because the time complexity of evaluating an elementary policy for a 1:$|L_j|$-system is $O(|L_j|)$ as pointed out above, the time complexity of evaluating the EMS-policy Π is $O(\sum |L_j|) = O(n)$ as $|L_j| \leq |N_j|$ and $\sum |N_j| = n$.

The last theorem of this section shows that the class of elementary policies always contains an optimal module-sequence policy.

Theorem 2.12. *There exists an EMS-policy that is optimal in the class $\mathcal{C}_E$ of elementary policies. Consequently,* $\max\{\mathbb{E}[f(\Pi)] \mid \Pi \in \mathcal{C}_E\} = \max\{\mathbb{E}[f(\Pi)] \mid \Pi \in \mathcal{C}_{EMS}\}$.

Proof. Consider an elementary policy $\Pi(\,\cdot\,; L)$ and assume $L = (L_i, \widetilde{L}, k_i, \widehat{L})$ with $L_i \subset N_i$, $\widetilde{L} \cap N_i = \emptyset$ and $k_i \in N_i$. We show that the expected profit does not decrease when the block L_i is moved just before job k_i; in other words, we prove that $\mathbb{E}[f(\Pi(\,\cdot\,; L'))] \geq \mathbb{E}[f(\Pi(\,\cdot\,; L))]$ with $L' = (\widetilde{L}, L_i, k_i, \widehat{L})$. The expected profit can be computed using Equation (2.2). If the elementary policy defined by job list L resp. L' is applied, we denote by S resp. S' the probability of project success, and by $R(k)$ resp. $R'(k)$ the probability of paying for job k. It is clear that the probability of project success is insensitive to the order in which the jobs are sequenced. It is also to be expected that the probability of achieving success for a job in the block L_i decreases when it is moved further in the list because the jobs in the block $\widetilde{L}$ now appear before block L_i and these may cause the project to fail before block L_i is reached. Remember that there are no jobs in block $\widetilde{L}$ of module i and also remember that all jobs of block L_i belong to module i. Therefore, the probability of paying for a job in $\widetilde{L}$ remains unchanged when block L_i is moved in the list. It is also clear that

the probability of paying for a job in block $\widehat{L}$ is unaffected by this action since the position of these jobs remain unchanged. The reader may verify this by computing these probabilities exactly with the recursive formulas (2.3)–(2.4), which results indeed in $S' = S$, $R'(k) = (1-\alpha)R(k)$ for $k \in L_i$, and $R'(k) = R(k)$ otherwise. The constant α is the probability that the project fails due to the processing of the jobs in block $\widetilde{L}$, and is given by:

$$\alpha = 1 - \prod_{\substack{j:N_j \cap \widehat{L} = \emptyset, \\ j \neq i}} \left(1 - \prod_{k \in \widetilde{L} \cap N_j} q_k \right).$$

Therefore, $R'(k) \leq R(k)$ for all k and the theorem follows. □

2.4 Algorithms

Two exact algorithms to solve problem MP1 are presented in this section. A dynamic-programming algorithm that finds a globally optimal policy is discussed in Section 2.4.1, followed by the description of a branch-and-bound algorithm that finds an optimal elementary policy in Section 2.4.2.

2.4.1 Dynamic programming

We describe a backward stochastic dynamic-programming (stochastic DP, SDP) recursion to find a globally optimal policy for MP1. The algorithm is loosely inspired by the DP in Kulkarni and Adlakha (1986), which is an exact method for deriving the distribution and moments of the earliest project completion time of Markovian PERT networks. At any decision moment t, the *status* of an activity is either *idle* or *redundant*. An activity is idle when it has not yet been started and success is not yet achieved for its module. An activity is redundant when it has been processed or when its module was successfully completed. Denote by Y the set of idle activities and by R the redundant activities. At any decision moment t, these sets constitute a partition of the job set: $N = Y(t) \cup R(t)$ and

$Y(t) \cap R(t) = \emptyset$. The *state* of the system corresponds with one choice for the status for each activity and is completely determined by Y because $R = N \backslash Y$.

A state $Y \in 2^N$ is called *feasible* when membership of Y implies membership of Y for all successor activities according to B^*. We let $\mathcal{S}$ represent the set of feasible states; $\mathcal{S}$ will also be called the *state space*. Formally, we have $Y \in \mathcal{S}$ if and only if $k \in Y$ implies $l \in Y$ for all $(k, l) \in B^*$. The dummy end job is always a member of Y.

For an MP1 instance $\mathcal{I}$ and a feasible system state Y, a smaller MP1 instance $\mathcal{I}(Y)$ is defined by removing all non-dummy redundant jobs in $N \backslash Y$ from the modular network of $\mathcal{I}$, removing empty modules, and also removing the corresponding elements from A and B_i $(i = 1, \ldots, m)$. The value function $G : \mathcal{S} \mapsto \mathbb{R}_+$ of the SDP recursion maps a feasible state Y onto the expected profit of an optimal policy for MP1 instance $\mathcal{I}(Y)$. We always have $G(Y) \geq 0$ because the empty policy is included in the policy class. The maximum expected profit of the initial MP1 instance $\mathcal{I}$ equals $G(N \backslash \{0\})$ because $\mathcal{I}(N \backslash \{0\}) = \mathcal{I}$. As initial boundary condition, we set $G(\{n+1\}) = V$. For $|Y| > 1$, we define the set of eligible activities $E(Y)$ as the subset of Y with all preceding jobs being redundant, i.e., $l \in E(Y)$ if and only if $l \in Y$, and $k \notin Y$ for all $(k, l) \in B^*$. The SDP relies on the following recurrence relation:

$$G(Y) = \max_{k \in E(Y)} \Big\{ 0 \,,\, p_k G(Y_{p,k}) + q_k H(Y_{q,k}) - c_k \Big\}, \quad \text{for all } Y \in \mathcal{S} \setminus \{\{n+1\}\} \tag{2.5}$$

$$G(\{n+1\}) = V \tag{2.6}$$

where $Y_{p,k} = Y \backslash N_{i_k}$, $Y_{q,k} = Y \backslash \{k\}$, and

$$H(Y_{q,k}) = \begin{cases} 0 & \text{if } Y \cap N_{i_k} = \{k\}, \\ G(Y_{q,k}) & \text{otherwise.} \end{cases}$$

An optimal policy Π^* can be extracted from the SDP by registering the jobs where the maxima are reached in (2.5). Concretely, if $\mathbf{x}$ is a realization,

$[\Pi^*(\mathbf{x})]_0$ is a job where the maximum is reached in (2.5) for state $Y^0 = N\backslash\{0\}$. Denote this job by k_1 and its module by i_1. If $x_{k_1} = 1$, we move to state $Y_p^1 = Y^0\backslash N_{i_1}$ and $[\Pi^*(\mathbf{x})]_1$ is a job where the recurrence relation reaches its maximum for state Y_p^1. If $x_{k_1} = 0$ and $|N_{i_1}| > 1$, we move to state $Y_q^1 = Y^0\backslash\{k_1\}$ and $[\Pi^*(\mathbf{x})]_1$ is a job where the recurrence relation reaches its maximum for state Y_q^1. If $|N_{i_1}| = 1$ the schedule ends ($|\Pi^*(\mathbf{x})| = 1$). Proceeding in this way, we can construct schedule $\Pi^*(\mathbf{x})$. The project is abandoned when the maximum over all eligible jobs is negative, which coincides with a value function of zero in (2.5).

Theorem 2.13. *Recurrence relation (2.5)–(2.6) finds a globally optimal policy.*

Proof. Let $\mathcal{I}$ be an arbitrary MP1 instance and let Π be a reasonable policy with decision-tree representation T and expected profit π. Denote the root node of T by k and the module containing job k by i. If job k is successful, we consider the instance $\mathcal{I}_p$ obtained by removing module i together with adjacent inter-modular precedence constraints from the modular network of instance $\mathcal{I}$. The subtree T_p of T emerging from the success edge of node k of T is a decision-tree representation of a policy Π_p for instance $\mathcal{I}_p$ with expected profit π_p. If job k fails, we consider the instance $\mathcal{I}_q$ obtained by removing job k from module i. Let T_q be the subtree emerging from the failure edge of node k of T. If k is the only job in module i then T_q consists of only one (leaf) node, labeled F. Otherwise, T_q is a tree representation of a policy Π_q for instance I_q with expected profit π_q. The expected profit of Π can be expressed as $\pi = p_k\pi_p + q_k\bar{\pi}_q - c_k$ with $\bar{\pi}_q = 0$ if $N_i = \{k\}$ and $\bar{\pi}_q = \pi_q$ otherwise. It follows that policy Π is optimal for instance $\mathcal{I}$ if the policies Π_p and Π_q are optimal for the smaller instances $\mathcal{I}_p$ and $\mathcal{I}_q$, respectively. □

Theorem 2.14. *The optimal policy generated by recurrence relation (2.5)–(2.6) is a dominant policy.*

Proof. Let Π^* be a policy generated by the SDP as explained above. To

show that Π^* is dominant, we choose two equivalent nodes k_1 and k_2 of the decision tree T^* of Π^*. The decisions made from these nodes are identical because they correspond to the same state in the recurrence relation. The state $Y(k)$ corresponding to a node k of T^* is given by the complement of $C_0(k) \cup \{N_i \,|\, N_i \cap C_1(k) \neq \emptyset\}$. From (E1) and (E2), it follows that $Y(k_1) = Y(k_2)$. □

For an efficient implementation of the DP recursion, we construct a total order relation $\leq_{\mathcal{S}}$ on the state space $\mathcal{S}$ that determines the order in which the states are evaluated. The relation is such that the value-function values in the right-hand side of (2.5) that are input to the computation of $G(Y)$ have already been computed beforehand. We observe that the corresponding states always have a cardinality less than $|Y|$. This implies that any ordering $\leq_{\mathcal{S}}$ respecting $Y_1 \leq_{\mathcal{S}} Y_2 \Leftrightarrow |Y_1| \leq |Y_2|$ satisfies our needs. To find a suitable order, the approach taken in Creemers et al. (2010) is followed by partitioning the state space according to the so-called rank of inclusion-maximal antichains[2] of the induced network (N, B^*). An inclusion-maximal antichain is also referred to as a *uniformly directed cut* (UDC). We denote the set of UDCs by $\mathcal{U}$. The UDCs of the MP1 instance depicted in Figure 2.1 are $U_0 = \{0\}$, $U_1 = \{1, 3\}$, $U_2 = \{2, 3\}$, $U_3 = \{4, 5\}$, and $U_4 = \{6\}$. Let U be a UDC and denote by $N(U)$ the subset of N containing the successor jobs of U; in other words,

$$N(U) = \{l \in N \,|\, \exists\, k \in U : (k, l) \in B^*\}.$$

The *rank* r of a UDC is the number of predecessor activities in the induced network, i.e., $r(U) = |N \backslash (N(U) \cup U)|$. For the example, we have $N(U_0) = \{1, 2, 3, 4, 5, 6\}$, $N(U_1) = \{2, 4, 5, 6\}$, $N(U_2) = \{4, 5, 6\}$, $N(U_3) = \{6\}$, $N(U_4) = \emptyset$, and $r(U_0) = 0$, $r(U_1) = 1$, $r(U_2) = 2$, $r(U_3) = 4$, $r(U_4) = 6$.

[2] An antichain S of a poset (V, O) is a subset of V such that any two elements of the subset are incomparable; thus for any $(i, j) \in O$ at least one of the elements i and j are no element of S. An antichain S is incusion-maximal if there is no element $i \in V \setminus S$ for which $S \cup \{i\}$ is an antichain.

A set of systems states (which are subsets of N) is associated with each $U \in \mathcal{U}$ and is denoted by $\sigma(U)$. The set $\sigma(U)$ contains all feasible system states for which all successor activities of U together with a non-empty subset of the activities in U are idle. In order to avoid the same states to appear for different UDCs, every successor activity of U needs at least one idle predecessor activity, such that none of them are eligible. Formally, a set $Y \subset N$ belongs to $\sigma(U)$ if we can write $Y = N(U) \cup U'$ with U' a non-empty subset of U such that the set of eligible activities $E(Y) \subset U$. Clearly, $N(U) \cup U$ always belongs to $\sigma(U)$ because $E(N(U) \cup U) = U$. For the example of Figure 2.1 the states corresponding to the UDC $U_2 = \{2, 3\}$ are $\{2, 4, 5, 6\}$, $\{3, 4, 5, 6\}$, and $\{2, 3, 4, 5, 6\}$. For $U_1 = \{1, 3\}$ we only have two states, namely $\{1, 2, 4, 5, 6\}$ and $\{1, 2, 3, 4, 5, 6\}$; state $\{2, 3, 4, 5, 6\}$ is not included in $\sigma(U_1)$ because the only predecessor of activity 2 is not idle, such that activity $2 \notin U_1$ is eligible. In the following theorem, we summarize results obtained in Creemers et al. (2010):

Theorem 2.15. *Let U and U' be two UDCs and let $Y \in \sigma(U)$, $Y' \in \sigma(U')$. We have*

1. *$\{\sigma(U) \,|\, U \in \mathcal{U}\}$ is a partition of $\mathcal{S}$.*
2. *Assume Y is in the left-hand side, and Y' is in the right-hand side of (2.5). If $U' \neq U$ then $r(U') > r(U)$.*
3. *If $r(U') > r(U)$ then $|Y'| < |Y|$.*

From Theorem 2.15 we infer an appropriate total order on the state space $\mathcal{S}$ by enumerating the parts $\sigma(U)$ of the partition in non-increasing rank order and the states in a given set $\sigma(U)$ in non-decreasing cardinality order. This leads to the following steps in computing the recurrence relation for the example of Figure 2.1:

- $\sigma(U_4) = \{Y_0\}$ with $Y_0 = \{6\}$; $G(Y_0) = V$.
- $\sigma(U_3) = \{Y_1, Y_2, Y_3\}$ with $Y_1 = \{4, 6\}$, $Y_2 = \{5, 6\}$, $Y_3 = \{4, 5, 6\}$;

$$G(Y_1) = \max\{0, p_4 G(Y_0) - c_4\},$$
$$G(Y_2) = \max\{0, p_5 G(Y_0) - c_5\},$$
$$G(Y_3) = \max\{0, p_4 G(Y_0) + q_4 G(Y_2) - c_4, p_5 G(Y_0) + q_5 G(Y_1) - c_5\}.$$

- $\sigma(U_2) = \{Y_4, Y_5, Y_6\}$ with $Y_4 = \{3, 4, 5, 6\}$, $Y_5 = \{2, 4, 5, 6\}$, and $Y_6 = \{2, 3, 4, 5, 6\}$;

$$G(Y_4) = \max\{0, p_3 G(Y_3) - c_3\},$$
$$G(Y_5) = \max\{0, p_2 G(Y_3) - c_2\},$$
$$G(Y_6) = \max\{0, p_2 G(Y_4) - c_2, p_3 G(Y_5) - c_3\}.$$

- $\sigma(U_1) = \{Y_7, Y_8\}$ with $Y_7 = \{1, 2, 4, 5, 6\}$, $Y_8 = \{1, 2, 3, 4, 5, 6\}$;

$$G(Y_7) = \max\{0, p_1 G(Y_3) + q_1 G(Y_5) - c_1\},$$
$$G(Y_8) = \max\{0, p_1 G(Y_4) + q_1 G(Y_6) - c_1, p_3 G(Y_7) - c_3\}.$$

The maximum expected profit is $G(Y_8)$.

2.4.2 Branch and bound

In this section, we develop an algorithm to optimize over the class $\mathcal{C}_{EMS}$ of elementary module-sequence policies. An optimal policy in this class is also optimal in the superclass $\mathcal{C}_E$ (Theorem 2.12). Adapting the SDP described in the previous section does not seem to be straightforward. To see this, reconsider the argument used in the proof of Theorem 2.13: a policy Π of an instance $\mathcal{I}$ can be decomposed in two policies Π_p resp. Π_q for smaller instances $\mathcal{I}_p$ resp. $\mathcal{I}_q$. Unfortunately, policy Π is not necessarily elementary even if policies Π_p and Π_q are elementary, making the recurrence relation (2.5) invalid for elementary policies. To further illustrate this, consider the instance and policy Π^* depicted in Figure 2.3. This policy is not elementary despite the fact that Π_p^* and Π_q^* are elementary (and module-sequence) with corresponding order lists $(3, 4)$ and $(3, 2)$ respectively. The fact that an elementary policy is representable by a job list makes the solution space $\mathcal{C}_E$ very suitable for implicit enumeration by

means of a branch-and-bound (B&B) algorithm, however, and this section is devoted to the development of such an algorithm.

Branching strategy

We devise a branching tree that enumerates the elements of the class $\mathcal{C}_{EMS}$. A node η in the tree is labeled with a job $k(\eta)$ and represents a partial order list $PL(\eta)$, determined by the node labels of the unique path in the search tree starting from the root node and ending in the node labeled $k(\eta)$. The root node at level 0 is labeled with the dummy start job 0 and every leaf node is labeled with the dummy end job $n+1$. Any partial order list $PL(\eta)$ must be a *compatible* partial list, that is, it satisfies conditions (C2)-(C4) of a compatible list defined in Section 2.2.2. Denote by $M_u(\eta)$ resp. $M_s(\eta)$ the set of unstarted resp. already started modules, i.e., $M_u(\eta) = \{i \in \{1,\ldots,m\} \mid N_i \cap PL(\eta) = \emptyset\}$ and $M_s(\eta) = \{1,\ldots,m\} \backslash M_u(\eta)$. Consider a node η labeled with a job k from a module i. A node η' labeled with a job k' from a module i' is a child node of node η if conditions (P1)–(P3) below hold.

(P1) $k' \notin PL(\eta)$.

(P2) $(PL(\eta), k')$ is a compatible partial list.

(P3) $i' \notin M_s(\eta) \backslash \{i\}$.

The collection of all the leaf nodes corresponds to the collection of all compatible job lists for which jobs belonging to the same module are consecutive, and thus belong to the class $\mathcal{C}_{EMS}$.

Figure 2.6 shows the branching tree of an MP1 instance with the modular network of Figure 2.1(a). Removal of condition (P3) defines a branching tree for class $\mathcal{C}_E$, where the consecutive execution of jobs from the same module is not enforced. For the example network, this would mean that node B of the branching tree depicted in Figure 2.6 would have a third child, labeled with job 2.

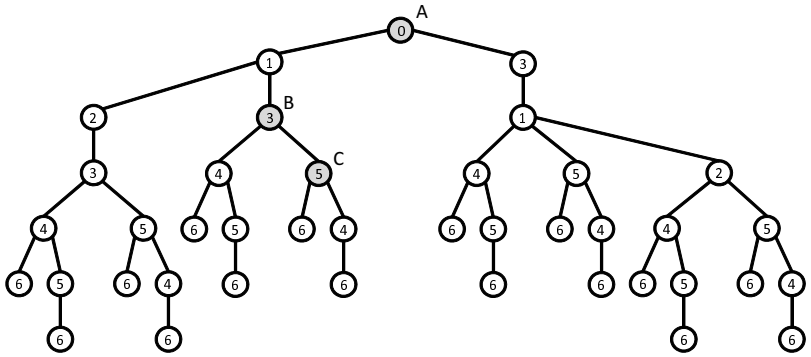


Figure 2.6: *Branching tree for an MP1 instance with modular network as depicted in Figure 2.1(a)*

Upper bound

In our upper-bound computation we assume that the project is executed, thus we exclude the empty policy from the solution space. A negative upper bound at a given node is automatically pruned by the zero global lower bound (see Section 2.4.2). Consequently, at least one job in each module needs to be processed (condition (C1)), in order to render the project potentially successful. Our upper bound $\bar{z}(\eta)$ at node η consists of a positive part $\overline{EP}(\eta)$ that overestimates the expected payoff and a negative part $\underline{EC}(\eta)$ that underestimates the expected cost, and exploits the knowledge of the thus-far constructed partial list $PL(\eta)$.

The expected payoff increases when more jobs are included in the list. Therefore, an overestimation of the expected payoff results from scheduling all jobs in all modules, unless adding a job to the list leads to an invalid order list. Denote by $M_1(\eta)$ the set of modules in which no new activities can be started due to condition (P3), i.e., $M_1(\eta) = M_s(\eta)\backslash\{i\}$, and gather the remaining modules in $M_2(\eta) = \{1,\ldots,m\}\backslash M_1(\eta)$. The first term of the upper bound is given by

$$\overline{EP}(\eta) = \left[\prod_{i\in M_1(\eta)}\Big(1 - \prod_{k\in N_i\cap PL(\eta)} q_k\Big)\right]\cdot\left[\prod_{i\in M_2(\eta)}\Big(1 - \prod_{k\in N_i} q_k\Big)\right]\cdot V. \quad (2.7)$$

The probability of success of a module in the set $M_1(\eta)$ can be computed exactly because they are already finished (no new activities can start in these modules). This corresponds with the first factor between square brackets in the estimation of the expected payoff (Equation 2.7). For the remaining modules (belonging to $M_2(\eta)$) new activities may be started in the future. We overestimate the probability of success of these modules by assuming that all the jobs of these modules are executed. This corresponds to the second factor between brackets in Equation 2.7.

Notice that $M_1(\eta)$ may hold modules containing a job for which addition to the partial order list leads to a violation of condition (C4). The bound described by Equation (2.7) remains valid if we optimize over the class $\mathcal{C}_E$, but the set of forbidden modules $M_1(\eta)$ will be smaller; in this case $M_1(\eta) = \{i \in \{1,\ldots,m\} \,|\, N_j \cap PL(\eta) \neq \emptyset \text{ for some } (i,j) \in A\}$. Consequently, the bound is stronger for class $\mathcal{C}_{EMS}$ than for class $\mathcal{C}_E$.

To define the total cost of the expected profit (the second term of the upper bound), we define N_i' as the subset of N_i holding the first jobs of a module: $N_i' = \{l \in N_i \,|\, \nexists k \in N_i : (k,l) \in B_i\}$. The second term of the upper bound can be written as a sum of two terms:

$$\begin{aligned} \underline{EC}(\eta) = & \sum_{i \in M_s(\eta)} \left[\sum_{k \in N_i \cap PL(\eta)} R(k) c_k \right] + \\ & \sum_{i \in M_u(\eta)} \left[\left(\prod_{j:(i,j) \notin A} P(j) \right) \min_{k \in N_i'} c_k \right] \end{aligned} \tag{2.8}$$

with

$$P(j) = \begin{cases} \min_{k \in N_j'} p_k & \text{if } j \in M_u(\eta), \\ 1 - \prod_{k \in N_j \cap PL(\eta)} q_k & \text{if } j \in M_s(\eta). \end{cases} \tag{2.9}$$

The first term in Equation (2.8) is the exact contribution to the expected cost of the activities that are already scheduled (i.e. part of the partial list $EP(\eta)$). The quantity $R(k)$ is the probability of paying for job k when $PL(\eta)$ is applied (in line with the definition in Section 2.3) under the condition that no further activities in the module of job k are processed apart

from those in $PL(\eta)$. These values can be computed using the recursion discussed in the proof of Theorem 2.11 applied to the instance obtained by removing jobs that are not in $PL(\eta)$.

The second term in Equation (2.8) underestimates the expected cost for the modules for which no activities are yet in the partial list. The expected cost increases when more jobs are in the list, so an underestimation includes only a single job for each module that has no job in the partial list. Clearly we must choose one with a minimum cost. The probability of paying for such a job decreases when more modules are scheduled before it, because they all need to be successful. Therefore, we consider all modules that are allowed to be scheduled before it. The underestimation of the probability of success of these modules j is denoted by $P(j)$ and is given by Equation 2.9. Notice that this probability can be computed exactly for those modules that are already started since no new activities can be started here (we optimize over *MS*-policies). To underestimate $P(j)$ for modules j that are not yet started, we choose the smallest probability of success among the jobs that are immediately available.

The upper bound at the root node η_0 is a global upper bound and is given by

$$\bar{z}(\eta_0) = \prod_{i=1}^{n} \left(1 - \prod_{k \in N_i} q_k\right) V - \sum_{i=1}^{m} \left(\prod_{j:(i,j)\notin A} \min_{k \in N'_j} p_k\right) \min_{l \in N'_i} c_l. \qquad (2.10)$$

The computation of the bound is illustrated for the nodes A, B, and C of the branching tree depicted in Figure 2.6. The global upper bound at the root node A is

$$\bar{z}(A) = (1 - q_1 q_2) p_3 (1 - q_4 q_5) V - (p_3 c_1 + p_1 c_3 + p_1 p_3 \min\{c_4, c_5\}).$$

For node B at level 2, we have a partial list $PL(B) = (0, 1, 3)$ and module sets $M_s(B) = \{1, 2\}$, $M_u(B) = \{3\}$, $M_1(B) = \{1\}$, and $M_2(B) = \{2, 3\}$. Since $R(1) = 1$ and $R(3) = p_1$, the local upper bound in B is

$$\bar{z}(B) = (1 - q_1) p_3 (1 - q_4 q_5) V - (c_1 + p_1 c_3 + p_1 p_3 \min\{c_4, c_5\}).$$

The bound would weaken if we optimized over $\mathcal{C}_E$ instead of over $\mathcal{C}_{EMS}$: then $M_1(B) = \emptyset$ and $M_2(B) = \{1, 2, 3\}$, and the factor $1 - q_1$ in $\overline{EP}(B)$ would change to $1 - q_1 q_2$. Finally, in node C at level 3, we have $PL(C) = (0, 1, 3, 5)$, $M_s(C) = \{1, 2, 3\}$, $M_u(C) = \emptyset$, $M_1(C) = \{1, 2\}$, $M_2(C) = \{3\}$, $R(1) = 1$, $R(3) = p_1$, and $R(5) = 1 - (q_1 + p_1 q_3) = p_1 p_3$, resulting in a local upper bound given by

$$\bar{z}(C) = p_1 p_3 (1 - q_4 q_5) V - (c_1 + p_1 c_3 + p_1 p_3 c_5).$$

Global lower bound

Since the empty policy belongs to $\mathcal{C}_{EMS}$, an initial global lower bound is $\underline{z} = 0$. In each leaf node η, the local upper bound described in the previous subsection is also exactly the expected profit of the elementary policy defined by the (full) order list $PL(\eta)$ and hence constitutes a global lower bound ($\underline{z} = \bar{z}(\eta)$).

2.5 Computational experiments

We have implemented the algorithms in C++ using Microsoft Visual Studio 2010. The experiments were run on a Dell Desktop E8500 Optiplex 760 with an Intel Core 2 Duo processor with clock rate of 3.16 GHz and 3.21 GB of RAM, equipped with Windows XP Professional Version 2002 Service Pack 3. All CPU times are expressed in seconds.

In the next section (Section 2.5.1), we describe how the set of test instances is put together. Then we discuss some implementation issues for the DP algorithm in Section 2.5.2, followed by a presentation of the computational results in Section 2.5.3.

2.5.1 Data generation

We have generated two data sets, which are available on-line[3]. The first set exclusively consists of n:n-systems, whereas the second contains general MP1 instances, possibly holding more than one activity per module. For the first data set, the B&B algorithm finds optimal elementary policies with the same objective values as the DP (in line with Theorem 2.3), motivating a direct comparison of the performance of our algorithms. The n:n-instances are created similarly to De Reyck and Leus (2008), using the random network generator RanGen (Demeulemeester et al., 2003) to build directed graphs for a given value of n and of the density of the network, which is measured by the *order strength* OS. The order strength is defined as the number of precedence-related activity pairs divided by the maximum possible number of such pairs, which is $\binom{n}{2} = n(n-1)/2$. We create 10 instances for each of the combinations of values for the parameters n and OS, with $n \in \{10k \,|\, k = 1, \ldots, 12\}$ and $OS \in \{0.4, 0.6, 0.8\}$. The second data set contains instances for the same combinations of values for n and for the OS of the induced network as for the n:n-instances, although the OS value is now only an approximation. A detailed description of the generation of the precedence constraints and the approximation of the OS of the induced network is described below.

In the generated data sets, the cost c_k of an activity k is a random variate of a discrete uniform distribution on the set $\{0, 1, \ldots, 50\}$, and the success probability p_k is an independent observation of a continuous uniform distribution on the interval $[0.8; 1]$. For the first data set with n:n-systems, the end-of-project payoff V is an integer randomly selected from the interval $[0.5a; 2a]$, with a the value of the payoff such that the elementary policy $\pi(\,\cdot\,; L)$ produced by Algorithm 2 has zero expected profit, i.e., $a = (1/a_n) \sum_{k=1}^{n} a_{k-1} c_{L(k)}$, with $a_0 = 1$ and $a_k = \prod_{l=1}^{k} p_{L(l)}$, $k = 1, \ldots, n$.

For the second data set, for given n and OS, we generate five instances with $m = \lceil n/4 \rceil$ non-dummy modules and another five instances with

[3] Available at http://www.econ.kuleuven.be/public/NDBAC96/MP1_instances.htm

Algorithm 2 Heuristic for n:n-systems

1: $L = \emptyset$; $E =$ set of non-dummy jobs without non-trivial predecessors
2: **while** $E \neq \emptyset$ **do**
3: Choose a job k^* from E such that $\forall k \in E \backslash \{k^*\}$ we have $c_{k^*}/q_{k^*} \leq c_k/q_k$
4: Add job k^* to the end of L
5: Update $E = E \backslash \{k^*\} \cup \{$jobs in $N \backslash (E \cup L \cup \{n+1\})$ with all predecessors in $L\}$
6: **end while**
7: Return L

Algorithm 3 Heuristic for general instances

1: **for** each module i **do**
2: Let $N_i' = \{k \in N_i \mid \nexists l \in N_i : (l,k) \in B_i\}$
3: Let $k_i = \arg\min\{c_l/p_l \mid l \in N_i'\}$
4: **end for**
5: Apply Algorithm 2 to the m:m-system obtained by removing all jobs from module i except for job k_i

$m = \lceil n/2 \rceil$ non-dummy modules. The module network consisting of the $m+2$ modules is generated with RanGen; a discussion of the determination of the order strength OS' of the module network is postponed to the next paragraph. Each module receives at least one activity, and the remaining $n - m$ activities are randomly allocated to the m modules. This results in a modular network with $B_i = \emptyset$ for all $i \in M$. If the order strength of the induced network is greater than or equal to OS, we stop. Otherwise, let $B_i' = \{(k,l) \,|\, k,l \in N_i,\, k < l\}$ and repetitively choose a precedence constraint (k,l) from $B' = \cup_{i \in M} B_i'$ at random and add it, together with any corresponding transitive elements, to the modular network until the order strength of the induced network exceeds or equals OS. At each step of this process, set B' is updated by removing the newly added elements. The costs and probabilities are generated in the same way as for the first data set. The payoff V is chosen randomly in $[0.5b;\, 2b]$, where b is the

value of the payoff such that the elementary policy $\pi(\,\cdot\,; L)$ produced by Algorithm 3 has zero expected profit.

The value of OS' is chosen such that a generated instance contains about half of the total number of possible precedence constraints within each module on average. To this end, let t be the numerator of the order strength OS', i.e., $t = OS'\binom{m}{2}$, and let $\bar{n} = n/m$ be the average number of jobs per module. In case all modules contained exactly $\bar{n}$ activities, the order strength of the induced network without any precedence constraints within a module would equal $t\bar{n}^2/\binom{n}{2}$. Furthermore, there are $m\binom{\bar{n}}{2}$ possible elements in B' such that aiming for half of the elements of B' leads to an instance with an order strength OS of its induced network equal to $\big(OS'\binom{m}{2}\bar{n}^2 + m\binom{\bar{n}}{2}/2\big)/\binom{n}{2}$. Solving this equation for OS' finally leads to the formula

$$OS' = \frac{m(n-1)\,OS - (n-m)/2}{n(m-1)}.$$

The actual number of jobs in a module can sometimes significantly differ from the average $\bar{n}$. Moreover, RanGen also approximates the order strength OS' of the generated networks. These uncertainties lead to a data set in which the total number of constraints within a module of the instances is nicely spread between zero and the maximum value, where the precedence network of the module is a chain.

2.5.2 Implementation issues for the DP algorithm

The SDP recursion uses a bitwise representation for a set of activities. A subset of n jobs is represented by an array of size $\lceil n/32 \rceil$ containing 32-bit integers. A job k is an element of the subset if and only if the k-th bit of the integer array is a 1-bit. This representation allows us to deal with memory efficiently. Moreover, binary set operations like the union, intersection and complement of two sets and adding or removing an element of a set can be implemented more efficiently.

To generate the UDCs, observe that the set $\mathcal{U}$ of inclusion-maximal an-

tichains of the induced network coincides with the set of maximal independent sets of the undirected graph with node set N and an edge $\{k, l\}$ between a pair of jobs $k, l \in N$ if either $(k, l) \in B^*$ or $(l, k) \in B^*$. The algorithm presented in Johnson et al. (1988) is implemented to generate all maximal independent sets in lexicographic order, with only polynomial delay between the output of two successive independent sets. It should be noted that the algorithm cannot be easily adapted for modifications of the lexicographic order requirement. In particular, the authors in Johnson et al. (1988) show that there is no polynomial-delay algorithm for generating all maximal independent sets in reverse lexicographic order, unless P=NP. During the generation process, the rank of the UDC is computed and UDCs are grouped based on their rank.

The greatest concern in the implementation of the SDP recursion is the memory management. Since memory and not computation time will turn out to be the bottleneck, it is very important to deal with memory efficiently. For instances with $n \leq 32$ it is possible to represent the value function with an array of size 2^n, in which each index of the array is one of the 2^n possible states. In practice, however, this implementation is only suitable for solving very small instances, since the size of the array is exponential in the number of jobs. Storing an array of size 2^{30}, for example, would require too much RAM even for an average recent computer. Luckily, $|\mathcal{S}|$ is often substantially less than 2^n, especially for dense precedence networks (both intermodular as well as intramodular), which creates opportunities for handling the memory more efficiently.

We use a *hash table* to tackle the memory issues, which is a data structure enabling fast storage and lookup of the data elements. Elements of a hash table are pairs consisting of a *key* and a *value*. A *hash function* maps a key to one of the possible table entries and determines the location to store and retrieve the associated value. When two different keys are mapped to the same table entry, a *collision* occurs, which can be resolved in several ways. For an introduction to hash tables, see for example Cormen et al. (1990); Knuth (1998); Standish (1995).

Algorithm 4 Insert new element $(Y, G(Y))$ in the hash table; assume k items are already stored

1: $i \leftarrow h(Y)$
2: $A_1(k) \leftarrow (Y, G(Y), A_2(i))$
3: $A_2(i) \leftarrow k$

Algorithm 5 Retrieve value function at state Y

1: $i \leftarrow A_2(h(Y))$
2: **if** $(A_1(i)(0) == Y)$ **then**
3: **return** $A_1(i)(1)$
4: **else**
5: $i \leftarrow A_1(i)(2)$
6: **end if**
7: **if** $(i == -1)$ **then**
8: item is not in list
9: **else**
10: Go to line 2
11: **end if**

The process of hashing can be divided in two steps. In a first step, the key Y is mapped to an integer $h(Y)$ by some hash function h. In the second step, the integer $h(Y)$ is mapped to a table entry (bucket) of the hash table. The second step ensures that a key is mapped to an integer in the range $[0, L-1]$, with L the length of the hash table. A hash function producing an integer x undergoes a modulo division by L, guaranteeing a valid table entry. Typically, L is chosen to be a large prime P or a power of two. By choosing the latter, i.e. $L = 2^k$, the slightly more expensive modulo operator can be avoided since the result of the modulo operator coincides with the first k bits of x.

In our SDP, a key-value pair takes the form $(Y, G(Y))$ with $Y \in \mathcal{S}$ a feasible state and $G(Y)$ the value function at state Y. Denote by h the hash function that maps a key Y onto a bucket $h(Y)$ in the range $[0, L-1]$.

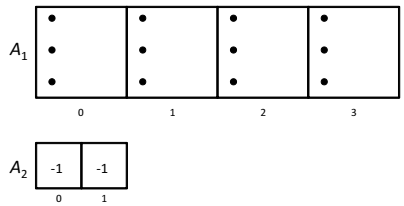
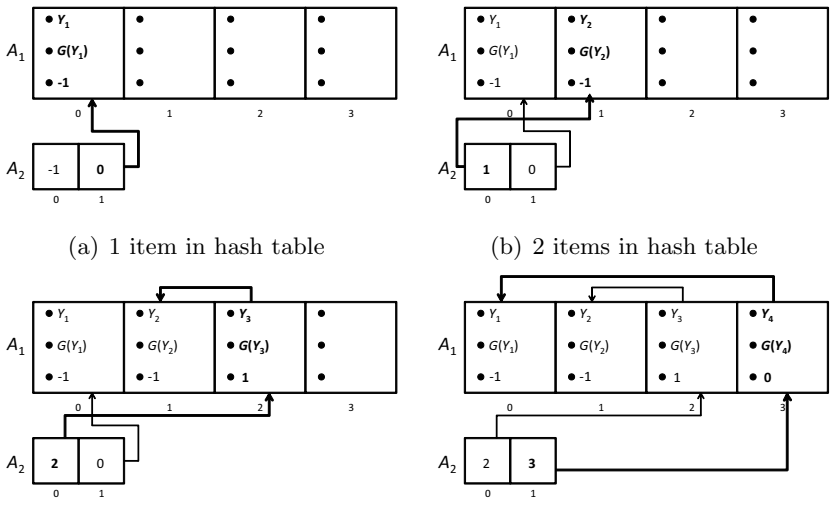


Figure 2.7: *No item in hash table*

Note that $|\mathcal{S}|$ can be computed exactly using the state-space partition in UDCs (Theorem 2.15), i.e. $|\mathcal{S}| = \sum_{U \in \mathcal{U}} |\{\sigma(U)|$. This allows for an efficient implementation of the hash table by means of two different arrays A_1 and A_2. The key-value pairs are stored in array A_1 from left to right, together with an integer that takes care of the collisions. This integer refers the previous element of A_1 with the same hash value. More concretely, an element $A_1(i) = (A_1(i)(0), A_1(i)(1), A_1(i)(2))$ is a triple $(Y, G(Y), k)$ for $i = 0, \ldots, \mathcal{S} - 1$, with k the largest integer smaller than i such that the key Y' of $A_1(k)$ maps to the same table entry, i.e. $h(Y) = h(Y')$. If no such integer exists, k equals -1. In this way, the keys of A_1 that map to the same bucket entry form a linked list. The linked list evolves from right to left and stops at an element $A_1(i)$ with $A_1(i)(2) = -1$. The array A_2 is of size L and $A_2(i)$ is the index of array A_1 that holds the head of the linked list containing the elements hashed onto table entry i for $i = 0, \ldots, L - 1$. If this list is empty we set $A_1(i) = -1$. Algorithm 4 shows how a new element is added to the hash table when k element are already present. Note that elements are added to the front instead of to the end of the linked list. Algorithm 5 describes how to retrieve the value function of a key stored in the table.

To illustrate this, consider an example involving 4 element $(Y_i, G(Y_i)), i = 1, 2, 3, 4$. Assume $L = 2$ and $h(Y_1) = h(Y_4) = 1$, $h(Y_2) = h(Y_3) = 0$. Figure 2.7 shows the situation before any element is in the list. Figure 2.8 illustrates the status of the lists A_1 and A_2 each time an item is added. At the end we have two linked lists: $3 \to 2$ resp. $4 \to 1$ corresponding to bucket entry 0 resp. 1.



(a) 1 item in hash table

(b) 2 items in hash table

(c) 3 items in hash table

(d) 4 items in hash table

Figure 2.8: *Status of hash table when new item i is added to the list, $i = 1, 2, 3, 4$*

Adding to the front of the linked list has two main advantages compared to adding to the end of the list. First of all, we can immediately add a new element to the hash table without having to go through the entire linked list (Algorithm 4). To retrieve an element, the linked list is scanned from left to right (Algorithm 5). Since more recently added elements are more likely to appear in further calculations, adding to the front of the linked list will retrieve elements faster.

To accomplish the first step of hashing a key to an integer value, remember from the beginning of this section that a key Y is represented by an integer array $Y = (y_0, \ldots, y_l)$ of size $l + 1 = \lceil n/32 \rceil$. Since a simple hash function of the form $\sum_{i=0}^{l} y_i$ results in equal hash values for different permutations of the components of Y, a multiplication with a position-dependent power of some constant c is performed, i.e. $h_0(Y) := \sum_{i=0}^{l} y_i c^i$. A large prime number is chosen for the constant c, which is a common hash function mainly used for string keys (Knuth, 1998).

Figure 2.9 shows the performance of h_0 on an n:n-instance with $n = 60$

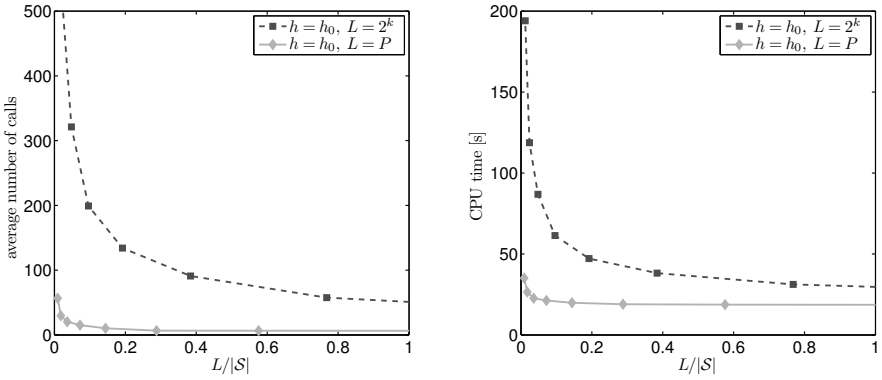


Figure 2.9: *Influence of the hash-table size on the performance of the hash table*

and $OS = 0.4$ [4], dependent on the table size L. The size of the hash table is scaled by the total number of elements in the list: the horizontal axis shows $L/|\mathcal{S}|$. The state space for the example instance is quite large, namely $|\mathcal{S}| = 10\,924\,600$. The left plot shows the average number of calls needed to retrieve an element from the hash table on the vertical axis, while the right plot depicts the CPU time. It is clear that choosing a prime number for the table size (diamond, light gray) is much better than a power of two (square, dark gray). The reason for this is probably that h_0 is actually a rather poor hash function, and simply dropping the $32 - k$ most significant bits results in too many collisions. The modulo operator with a prime number, on the other hand, if not too close to a power of two, will improve the randomness and is in fact a well known hash function for integer keys, known under the name *division method* (Cormen et al., 1990). Note that for the choice of the hash-table primes, a prime number is chosen close to the middle of the intervals $(2^k, 2^{k+1})$. The prime numbers should be as far as possible from a power of two to avoid clustering (Cormen et al., 1990; Knuth, 1998).

The quality of the hash function h_0 can be significantly improved when it is followed by a good integer hash function. Integer hash functions expect

[4] Available at http://www.econ.kuleuven.be/public/NDBAC96/MP1_instances.htm (filename *s_n60_os4_1*)

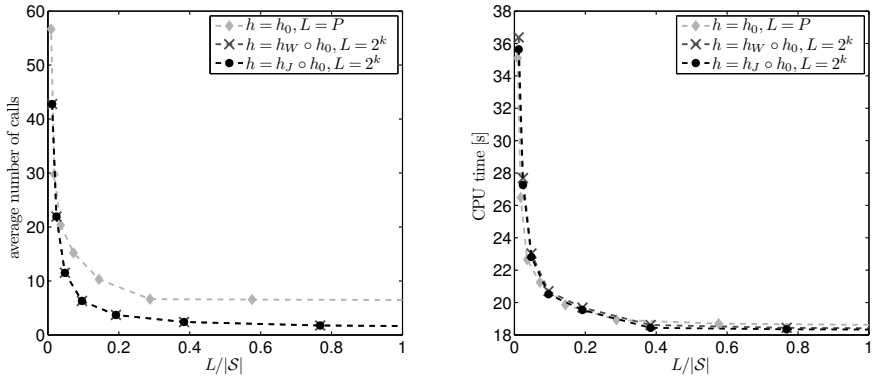


Figure 2.10: *Effect of the composition of h_0 with an integer hash function*

an integer key and manipulate the bits of the key to produce a randomized integer where the probability of a 1-bit or a 0-bit is equally likely for all the 32 bits of the integer. Good integer hash functions strive for a change in as many bits as possible when two different keys only differ in a single or a few bits. From the hash functions known from literature, Wang's "32-bit Mix Function" h_W and Jenkins' 32-bit integer hash function h_J perform the best (Wang, 2007). The left graph of Figure 2.10 shows that the quality of hash function h_0 (with prime hash-table sizes) improves after applying h_W or h_J (with power of two hash table sizes). The average number of calls to retrieve an element can be reduced from about 6.5 to 1.7 for a sufficiently large table size. The right graph indicates that the gain in CPU time is only minor, however, which is probably due to the increase in complexity of the hash function. Note that h_J is slightly better than h_W, both in quality and in efficiency. The difference in quality is very small and not even visible in the left picture.

We have observed a similar behavior for the other analyzed MP1 instances. Based on this analysis, we have opted for hash function $h_J \circ h_0$ as final implementation for solving the instances of the two data sets. Figure 2.10 suggests that a table size at least half the size of the state space is sufficient, since the improvement for larger sizes is only minor. The size of the hash table is set as the largest power of two smaller than $|\mathcal{S}|$, and

is thus between $|\mathcal{S}|/2$ and $|\mathcal{S}|$. Note that although the average number of calls is monotonically decreasing when L goes up (the quality of hash table improves with increasing L), this is not true for the CPU time: the efficiency of hash table can decrease when L is overly large. The largest table size that we have been able to implement is $L = 2^{28}$, which is about 25 times the size of the state space of the test instance. Compared to a good choice of the hash-table size, which is $L = 2^{23}$, the average number of calls drops to 1.1, but the CPU time increases by more than a second. A possible explanation for this might by the increase in cache misses since blocks of memory will contain more redundant lines holding unused hash table entries (Oliveira and Stewart, 2006).

2.5.3 Computational results

Table 2.2 reports the average CPU time of the algorithms for the first data set, which contains only n:n-systems. The number of instances solved until optimality (out of 10) is displayed together with the average CPU time. All averages reported in this section only pertain to the instances solved to optimality. The DP solves instances with 120 jobs and high OS (0.8) in less than a half second. When the order strength decreases, the state space grows (see Table 2.4(a)) and the instances become harder to solve. For $OS = 0.6$, we can solve instances with up to 100 jobs and more than 23 million states in less than one minute on average. Larger instances cannot be solved because the computer runs out of memory. If OS is further lowered to 0.4, the DP solves instances with $n = 60$ in a half minute. For larger networks, we again encounter memory problems. We conclude that the instances become harder when the density of the networks decreases, and that overall the DP solves quite large instances (with a very large state space) in little time. A possible reason for the former observation is the fact that the state space grows dramatically when the order strength decreases (Table 2.4(a)). We remark that the number of UDCs is not necessarily a good indicator for the hardness of an instance (not shown in a table). For the extreme case of an MP1 instance without precedence constraints

($OS = 0$), for example, we have only one UDC, but the size of the state space is maximal (2^n).

A time limit of 30 minutes is imposed when running the B&B algorithm. The results are also summarized in Table 2.2. We conclude that the B&B is far slower than the DP; each unsolved instance is interrupted due to the time limit, there are no memory problems. This also means more instances will likely be solved by simply increasing the time limit. For instances with $OS = 0.8$, the algorithm can solve all the instances of the dataset with at most 50 activities. When the number of activities increases to 60, the B&B can still solve 9 out of 10 instances (for $OS = 0.8$) within the time limit. For order strengths of 0.6 and 0.4, we can solve all instances with at most 40 and 30 activities, respectively. For larger instances, the B&B can only solve a few instances in each instance group. It is interesting to note that these larger solved instances, all have the empty policy (with expected cost zero) as the optimal elementary policy. Instances for which the empty policy is optimal are often easier to solve because a negative bound is more likely to be obtained, resulting in the direct pruning of that part of the search tree. Table 2.4(b) shows the average number of nodes (rounded to the nearest integer) explored by the B&B algorithm. This number grows rapidly up to the value of n for which the number of instances solved is less than 50%. The instances that are solved for larger n are those with optimal cost zero, and correspond to a lower number of visited nodes.

In Table 2.3, the running times and the number of solved instances are shown for the second data set, which contains general MP1 instances. The DP is again by far more efficient than the B&B. Compared to the first data set, the difficulty of the instances for a fixed value of n and OS is significantly more heterogenous. A possible cause may be that the order strength of the induced network does not accurately capture the complexity of an instance anymore. The precedence constraints between modules may only have a small influence on the order strength of the induced network, but they can have an important impact on the size of the state space and thus on the difficulty of the instance. If multiple modules contain

only few precedence constraints, the size of the state space tends to grow significantly. This heterogeneity in the network structure for a fixed value of n and OS may be one reason why some instances are solved by the B&B and others are not (Table 2.3). It should also be noted that the quality of the bound of the B&B depends on the values of the activity costs and probabilities, and the payoff. Instances for which the empty policy is optimal, for example, are often easier to solve because a negative bound is more likely to be obtained, resulting in the direct pruning of that part of the search tree. This sensitivity may explain why some instances in the same cell of Table 2.2 are solved by the B&B, whereas others are not. The same explanation may hold for the unsolved instances of Table 2.3 with few jobs per module (in which case the density of the networks within a module does not play an important role).

The B&B algorithm solves 85 instances to optimality within the time limit. Only five instances out of 85 have a relative optimality gap different from 0%; the average optimality gap of the elementary policies for these five instances is 0.16%. We conclude that, on average, the quality of an optimal elementary policy is quite close to that of a globally optimal policy.

Table 2.2: *Comparison of average CPU times (in seconds) and number of solved instances (out of 10) between DP and B&B for the data set containing only n:n-instances*

	$OS = 0.8$				$OS = 0.6$				$OS = 0.4$			
	DP		B&B		DP		B&B		DP		B&B	
n	CPU (s)	#	CPU (s)	#	CPU (s)	#	CPU (s)	#	CPU (s)	#	CPU (s)	#
10	0.00	10	0.00	10	0.00	10	0.00	10	0.00	10	0.00	10
20	0.00	10	0.00	10	0.00	10	0.01	10	0.00	10	0.04	10
30	0.00	10	0.08	10	0.00	10	2.20	10	0.02	10	34.91	10
40	0.00	10	2.68	10	0.01	10	335.18	10	0.23	10	552.38	4
50	0.00	10	74.97	10	0.04	10	1.02	3	2.39	10	1.20	2
60	0.00	10	463.52	9	0.21	10	0.14	3	30.73	10	122.62	4
70	0.01	10	5.78	4	1.03	10	0.06	2		0	0.78	1
80	0.02	10	0.05	1	3.99	10	29.72	2		0	15.17	1
90	0.05	10	12.30	1	15.35	10	277.26	2		0	23.49	2
100	0.09	10	3.61	2	56.62	8	149.66	3		0	479.89	2
110	0.21	10	8.69	1		0	1204.61	1		0		0
120	0.46	10	28.04	2		0	43.65	2		0	705.13	2

Table 2.3: *Comparison of average CPU times (in seconds) and number of solved instances (out of 10) between DP and B&B for the second data set (general MP1 instances)*

	$OS = 0.8$				$OS = 0.6$				$OS = 0.4$			
	DP		B&B		DP		B&B		DP		B&B	
n	CPU (s)	#	CPU (s)	#	CPU (s)	#	CPU (s)	#	CPU (s)	#	CPU (s)	#
10	0.00	10	0.00	10	0.00	10	0.00	10	0.00	10	0.00	10
20	0.00	10	0.04	10	0.00	10	0.92	10	0.00	10	36.85	10
30	0.00	10	23.53	10	0.00	10	161.68	5	0.01	10	350.14	3
40	0.00	10	734.52	6	0.02	10	107.61	1	1.23	10		0
50	0.00	10		0	0.04	10		0	5.95	9		0
60	0.02	10		0	1.54	10		0	23.6	8		0
70	0.06	10		0	3.27	10		0	41.9	6		0
80	0.07	10		0	16.65	10		0	403.9	4		0
90	0.19	10		0	24.6	9		0	2041.1	2		0
100	0.21	10		0	21.1	4		0		0		0
110	1.09	10		0	128.3	4		0		0		0
120	1.48	10		0	363.6	4		0		0		0

Table 2.4: *Additional indicators of the computational effort of the two algorithms for each instance group specified by n and OS of the first data set (n:n-instances)*

(a) Average size of the state space in the DP

n	$OS = 0.8$	$OS = 0.6$	$OS = 0.4$
10	22	40	91
20	88	316	1557
30	254	2044	16284
40	689	9650	176200
50	1460	41983	1421452
60	3657	162351	14074728
70	8092	644306	
80	15569	2176367	
90	31474	7335005	
100	61867	23517689	
110	131360		
120	280685		

(b) Average number of visited nodes in the B&B tree

n	$OS = 0.8$	$OS = 0.6$	$OS = 0.4$
10	12	21	47
20	60	502	2347
30	2281	41834	611871
40	34514	2039755	3920018
50	558017	15958	9401
60	1758477	1503	550499
70	32146	463	5103
80	505	160839	58133
90	72800	977694	74814
100	17915	370554	1098580
110	28352	3216533	
120	78076	69796	1133910

Table 2.5: *Additional indicators of the computational effort of the two algorithms for each instance group specified by n and OS of the second data set (general MP1 instances)*

(a) Average size of the state space in the DP

n	$OS = 0.8$	$OS = 0.6$	$OS = 0.4$
10	21	44	72
20	80	325	1094
30	263	2905	8941
40	790	13452	622098
50	2587	19032	2486256
60	3955	736142	2217939
70	32972	1481591	3922089
80	18416	5792564	17851239
90	91720	7234182	13352239
100	116229	5457474	
110	494835	15210952	
120	595798	13552638	

(b) Average number of visited nodes in the B&B tree

n	$OS = 0.8$	$OS = 0.6$	$OS = 0.4$
10	47	138	279
20	2990	76063	2868970
30	1212525	7144582	24123319
40	26224724	5092757	

2.6 Summary, conclusions, and further research

This chapter studies a model for scheduling R&D projects to maximize the expected profit when activities have a possibility of failure. The model extends earlier work by introducing a two-layered network structure, where activities are grouped in modules such that individual activity default does not necessarily imply an overall project failure. Processing an activity implies a negative cash flow (cost). Only when the project succeeds, a positive cash flow (payoff) is obtained. Activities are scheduled on a single machine. However, this is not a further limitation but rather a consequence of the assumption of time independent cash flows. In this setting, a solution is a policy, which can be naturally described by a binary decision tree.

Multiple policy classes are proposed and the relations among the classes is examined. Elementary policies are determined by a list of activities and have a compact representation, in contrast with more general classes of policies. We show that elementary policies are globally optimal for a number of specific network structures, but not in general. We also prove that it is sufficient to execute the jobs module by module when the solution space is restricted to elementary policies. Although the general scheduling problem is NP-hard, some special cases are shown to be polynomially solvable.

A backward stochastic dynamic-programming algorithm is developed to produce globally optimal policies. The algorithm is efficient and solves large instances with more than 100 jobs, with the performance dependent on the density of the network. The bottleneck of the algorithm is the memory rather than the computation time. Because an adaptation of the dynamic-programming algorithm to elementary policies is not straightforward, a branch-and-bound algorithm is proposed to find an optimal elementary policy. Notwithstanding the smaller solution space, in our implementations the optimization over elementary policies is significantly more time-consuming than the dynamic program, making the computation time the bottleneck of this algorithm. On average, the quality of an optimal

elementary policy is quite close to that of a globally optimal policy. There are instances nevertheless where a project would not be executed (has a zero objective value) when execution is restricted to elementary policies, whereas a globally optimal policy with a strictly positive objective value exists.

When the network density is low and the number of jobs is too high, our exact algorithms either run out of memory (DP) or become very slow (B&B). In order to obtain a good solution to these difficult instances, further research is needed on the development of heuristics requiring little time and memory that produce near-optimal policies, which is exactly the topic of Chapter 3. Another valid research question is whether a more efficient exact solution procedure is achievable for elementary policies. As a more fundamental extension, we distinguish especially the incorporation into the problem statement of the time value of money by means of discounting. In this case, adhering to the simple resource structure that is studied in this thesis (a single machine) may no longer be advisable, and other scheduling policies that allow for parallel processing of activities will probably lead to better results.

Appendix: verification of observation 3

We examine the hypothesis that for the MP1 instance presented in Section 2.2 the non-elementary policy Π^* as described by Figure 2.3(b) would yield a higher expected total profit than any elementary policy. For our analysis, we divide all 56 elementary policies[5] that select at least one job into four classes according to the first job to be executed. Policies in class 1 execute job 1 in the first place, and similarly for classes 2, 3 and 4. Below, we derive a number of conditions on the parameters c, V and p for policy Π^* to be strictly better than the elementary policies. One additional requirement is that the policy have a strictly positive expected profit, i.e.

[5] $56 = \binom{2}{1}\binom{2}{1}2 + \binom{4}{3}3! + \binom{4}{4}4!$

Π^* is strictly better than the trivial (elementary) policy of abandoning the project immediately (represented by the empty list).

Comparison with class 1

We choose $p_1 = p_2 = p_3 = p_4 = \frac{1}{2}$ and we also impose

$$c_3 < c_2 \tag{2.11}$$

and

$$c_3 < c_4. \tag{2.12}$$

We will represent an elementary policy defined by job list $(abcd)$ as Π_{abcd}. Below, we establish conditions under which Π^* is better than each of the 14 policies in class 1.

(i) Inequality $\mathbb{E}[f(\Pi^*)] > \mathbb{E}[f(\Pi_{1234})]$ holds if $-c_3 - p_3c_2 + p_2p_3V > -c_2 - p_2c_3 + p_2p_3V + p_2q_3p_4V - q_3p_2c_4$, which leads to

$$c_2 + \frac{1}{2}c_4 > c_3 + \frac{1}{4}V. \tag{2.13}$$

(ii) $\mathbb{E}[f(\Pi^*)] > \mathbb{E}[f(\Pi_{1243})]$ due to our findings under (i) and the fact that $\mathbb{E}[f(\Pi_{1243})] < \mathbb{E}[f(\Pi_{1234})]$. The latter inequality is due to Equation (2.12).

(iii) $\mathbb{E}[f(\Pi^*)] > \mathbb{E}[f(\Pi_{1324})]$ if $0 > -c_2 - p_2c_4 + p_2p_4V$, which corresponds to

$$V < 4c_2 + 2c_4. \tag{2.14}$$

(iv) $\mathbb{E}[f(\Pi^*)] > \mathbb{E}[f(\Pi_{1342})]$ if $0 > -c_4 - p_4c_2 + p_2p_4V$, or

$$V < 4c_4 + 2c_2. \tag{2.15}$$

(v) $\mathbb{E}[f(\Pi^*)] > \mathbb{E}[f(\Pi_{134})]$ when $-c_3 - p_3c_2 + p_2p_3V > 0$, or

$$V > 4c_3 + 2c_2. \tag{2.16}$$

Table 2.6: *Comparison between policies* Π_{1432} *and* Π^*.

$\mathbb{E}[f(\Pi_{1432})]$	$\mathbb{E}[f(\Pi^*)]$	inequalities
$-c_1 - \frac{1}{8}(4c_4 + 2c_2 - V)$	$-c_1$	(2.15)
$\frac{1}{16}(V - 4c_3 - 2c_2)$	$\frac{1}{8}(V - 4c_3 - 2c_2)$	(2.16)
$-\frac{1}{4}(2c_4 + c_3) + \frac{3}{8}V$	$-\frac{1}{4}(2c_3 + c_4) + \frac{3}{8}V$	(2.12)

(vi) We compute

$$\mathbb{E}[f(\Pi_{1432})] = (9/16)V - (c_1 + (3/8)c_2 + (1/2)c_3 + c_4)$$
$$\mathbb{E}[f(\Pi^*)] = (1/2)V - (c_1 + (1/4)c_2 + c_3 + (1/4)c_4)$$

In Table 2.6, both policies' profits are written as a sum of three terms and each term in the first column is strictly smaller than the corresponding term in the second column due to the equation referred to in the third column. We conclude that $\mathbb{E}[f(\Pi_{1432})] < \mathbb{E}[f(\Pi^*)]$ if we assume that the inequalities (2.12), (2.15), and (2.16) hold.

(vii) From (2.11) and (vi), we have $\mathbb{E}[f(\Pi_{1423})] < \mathbb{E}[f(\Pi_{1432})] < \mathbb{E}[f(\Pi^*)]$.

(viii) $\mathbb{E}[f(\Pi^*)] > \mathbb{E}[f(\Pi_{132})]$ if $p_4V - c_4 > 0$, or equivalently if

$$V > 2c_4. \tag{2.17}$$

(ix) From (2.12) and (v), we have $\mathbb{E}[f(\Pi_{143})] < \mathbb{E}[f(\Pi_{134})] < \mathbb{E}[f(\Pi^*)]$.

(x) From (2.11) and (viii), we have $\mathbb{E}[f(\Pi_{123})] < \mathbb{E}[f(\Pi_{132})] < \mathbb{E}[f(\Pi^*)]$.

(xi) From (2.12) and (viii), we have $\mathbb{E}[f(\Pi_{142})] < \mathbb{E}[f(\Pi_{132})] < \mathbb{E}[f(\Pi^*)]$.

(xii) From (2.12) and (x), we have $\mathbb{E}[f(\Pi_{124})] < \mathbb{E}[f(\Pi_{123})] < \mathbb{E}[f(\Pi^*)]$.

(xiii) From (2.16) and (viii), we have $\mathbb{E}[f(\Pi_{13})] < \mathbb{E}[f(\Pi_{132})] < \mathbb{E}[f(\Pi^*)]$.

(xiv) From (2.12) and (xiii), we have $\mathbb{E}[f(\Pi_{14})] < \mathbb{E}[f(\Pi_{13})] < \mathbb{E}[f(\Pi^*)]$.

From the above we conclude that policy Π^* has an expected profit that is greater than the expected profit of any elementary policy of class 1 when inequalities (2.11)–(2.17) hold and all success probabilities are equal to fifty percent.

Comparison with classes 2, 3 and 4

For a policy of class 3, we look at the policy of class 1 obtained by interchanging job 3 by job 1 and job 2 by job 4 in the associated decision trees. This results in a one-to-one correspondence between the policies of class 1 and class 3 with equal corresponding expected profits if we further impose

$$c_1 = c_3, \tag{2.18}$$

$$c_2 = c_4. \tag{2.19}$$

For any elementary policy of class 2, we can look at a corresponding policy of class 1 by an interchange of jobs 1 and 2 in the associated decision trees. This policy is clearly better than the corresponding policy of class 2 because $c_1 < c_2$ follows from (2.18) and (2.11). We can develop a similar argument for class 4 by an interchange of jobs 4 and 3 and the result follows from (2.12).

Conclusion

Policy Π^* is a non-elementary policy for the example instance with an expected profit strictly better than any elementary policy if we can find values for c_i $(i = 1, 2, 3, 4)$ and V satisfying (2.11)–(2.19) and with positive expected profit (we have chosen $p_i = \frac{1}{2}$ for all i).

The choice $c_1 = c_3 = 1$ and $c_2 = c_4 = 3$ satisfies equations (2.11), (2.12), (2.18) and (2.19). Equations (2.13)–(2.17) impose $10 < V < 14$. If we select $V = 13$, for example, then $\mathbb{E}[f(\Pi^*)] = 3 > 0$. Optimal elementary policies, on the other hand, achieve an expected profit of $\mathbb{E}[\Pi(\cdot; L)] =$

$47/16 = 2.9375$, which is the case, for instance, for the elementary policy defined by job list $L = (1, 2, 3, 4)$.

Chapter 3

Heuristic algorithm for MP1

The branch-and-bound and dynamic-programming algorithms that were presented in Chapter 2 obtain exact solutions for small to medium-size instances of MP1. In this chapter, we propose a heuristic that can be used for large instances, or when instances are particularly difficult because of their characteristics, such as low network density. A comparison with the results of the exact algorithms learns that the heuristic finds good quality solutions while requiring only very limited computation time and computer memory.

This chapter is the result of a collaboration with M. Huysmans, dr. F. Talla Nobibon and prof. dr. R. Leus. A technical report appeared as Research Report KBI_1227 (Huysmans et al., 2012). An article containing the material of this chapter is submitted to Journal of Heuristics.

3.1 Introduction

When the network density is low and the number of jobs is too high, our exact algorithms either run out of memory (DP) or become very slow (B&B). In order to obtain a good solution to these difficult instances, further research is needed on the development of heuristics requiring little time and memory that produce near-optimal policies, which is exactly the topic of Chapter 3. It seems to be a natural choice to restrict to the class of elementary policies for the development of heuristics, rather than work with the far less convenient combinatorial structures that represent dominant policies (binary decision trees). Moreover, we conclude from our experiments in Chapter 2 that the average optimality gap is quite low (the average difference between globally optimal policies and optimal elementary policies was less than one hundredth of a percent). Based on Theorem 2.12 we can even restrict ourselves to lists that schedule jobs module per module. In this chapter, we therefore focus on the class of elementary module-sequence policies (EMS-policies) as defined in the previous chapter (Section 2.2.3).

The two-stage approach described in Ben-Dov (1981b) for solving a related sequential testing problem constitutes a good starting point for developing a heuristic for problem MP1. The goal of the sequential testing policy is to find a testing policy which discovers the functioning of the overall system by sequentially testing the individual components of the system against a minimum expected inspection cost. In Section 3.2 we give more details on the description of the sequential testing problem, on the procedure proposed by Ben-Dov for the sequential testing problem for a special type of systems, and on the connection between this testing problem and problem MP1. In Section 3.3 it is explained that an adaptation of Ben-dov's procedure is necessary to ensure that all precedence constraints are respected. Moreover, the procedure does not take into account the possibility of discarding a job from the order list defining an EMS-policy. The latter aspect (job selection) might even be neglected in a first attempt: from the 85 instances that were solved by the B&B, only 14 instances have an optimal

EMS-policy whose list does not hold all jobs.

3.2 Sequential testing problems

In this section, we review a number of concepts and results from the sequential testing literature that can be transposed to the MP1 setting. The heuristic for MP1, to be described in Section 3.3, is based on an optimal procedure for testing so-called 'series-parallel' systems.

3.2.1 Link between testing and scheduling

A review of the literature on sequential testing can be found in Ünlüyurt (2004). In testing problems, one is concerned with a system function f rather than with the Boolean project success function as used in MP1, for a system consisting of a number of components. The system function f indicates the state of the system for every possible combination of states of the underlying components. The state of the components is given by a vector $\mathbf{x}$, with $x_i = 1$ if component i is working, and $x_i = 0$ if it is failing – we deliberately re-define some symbols, to underline their equivalence in the two settings (scheduling and testing); obviously, we equate activities in the scheduling problem with components in the testing problem. When the system is working, $f(\mathbf{x}) = 1$. When it is failing, $f(\mathbf{x}) = 0$. Each of the state variables x_i is the outcome of a Bernoulli variable X_i with probability of success p_i. The probability that the component fails is given by $q_i = 1 - p_i$.

The Boolean system function that has the closest ties to our MP1 scheduling problem is given by:

$$f(\mathbf{x}) = \bigwedge_{i \in M} \left(\bigvee_{k \in N_i} x_k \right). \tag{3.1}$$

This system function corresponds to a *series-parallel system of depth 2*. A *series-parallel system* (SPS) can be defined recursively as a system that

consists of a serial or parallel connection of subsystems that are themselves SPS (Ünlüyurt, 2004). The depth of an SPS is defined as $1+\max\{\text{depth of proper subsystem}\}$. For example, (simple) serial and parallel systems can be thought of as SPSs of depth 1, with system function $f(\mathbf{x}) = \wedge_i x_i$ and $f(\mathbf{x}) = \vee_i x_i$, respectively. There are two types of SPSs of depth 2, which are named according to their global structure. A two-level SPS that is a serial connection (respectively parallel arrangement) of parallel (respectively serial) subsystems, is also simply called a series-parallel (respectively parallel-series) system (Ben-Dov, 1981b).

A solution to a testing instance is also a policy Π, which maps each state vector $\mathbf{x} \in \mathbb{B}^n$ to a schedule $\mathbf{s}$. A major difference between testing and scheduling, however, is the following: when testing, one will continue exploring new components until the actual state of the system is known with certainty. This means that, regardless of the order in which components are tested, the final output will always be the actual state of the system. In MP1, on the other hand, one can abandon the project prematurely, the result being that the project fails even though it may have been possible to complete it successfully had more activities been undertaken, but the remaining activities may simply be too costly compared to the expected payoff. As a consequence, a scheduling policy Π may fail to identify $f(\mathbf{x})$ for certain scenarios $\mathbf{x} \in \mathbb{B}^n$, so that $\varsigma(\mathbf{x}, \Pi(\mathbf{x})) = 0$ even though $f(\mathbf{x})$ may have been equal to 1.

The second difference between testing and scheduling is the objective function. When testing, the goal is to minimize the expected cost of discovering the state of the system. The cost function $G(\mathbf{s})$ for schedule $\mathbf{s}$ is given by $\sum_{i\in\mathbf{s}} c_i$. Depending on the context, this can represent a financial cost or the time needed to test the components, or another measure of testing effort. A generic statement of the sequential testing problem, in line with our problem statement in Section 2.1.2, is then:

$$\Pi^* = \arg\min_{\Pi\in\mathcal{T}} \mathbb{E}\Big[G\big(\Pi(\mathbf{X})\big)\Big], \tag{3.2}$$

where $\mathcal{T}$ is a given class of testing policies. When no precedence constraints

apply to the sequencing of the component tests and with a system function of the form (3.1), we have the unconstrained series-parallel sequential testing problem, denoted USPST for short.

The concept of an elementary policy can also be defined for testing problems. In light of the impossibility of early abandonment (discussed supra), however, the compatible ordering L defining an elementary policy now needs to be an ordering of the entire component set N. One minor modification can be made, namely that L needs to contain only the non-redundant components of the instance, where a component k is redundant if it has no influence on the system, that is $\forall \mathbf{x} \in \mathbb{B}^n$ with $x_k = 0$ we have $f(\mathbf{x}) = f(\mathbf{x} \vee e_k)$, with e_k the k-th unit vector of $\mathbb{R}^n$. Without loss of generality, in the following we will simply work with an ordering of N. We thus conclude that the class $\mathcal{C}_E$ of elementary scheduling policies contains the class $\mathcal{T}_E$ of elementary testing policies, if the same parameters are considered.

3.2.2 Optimality results for specific sequential testing problems

In this section, we discuss a number of sequential testing problems without precedence constraints for which elementary policies are globally optimal. For a simple series system, the order $(1, 2, \ldots, n)$ is optimal if and only if $\frac{c_1}{q_1} \leq \frac{c_2}{q_2} \leq \ldots \leq \frac{c_n}{q_n}$. The proof relies on a straightforward interchange argument, and has been provided by Mitten in 1960 and by Butterworth in 1972; for discussions of this result, see for example Ben-Dov (1981b) and Ünlüyurt (2004). This result can be transferred to MP1: an elementary policy defined by the above ordering is optimal for an MP1 instance with each $|N_i| = 1$ and no precedence constraints (at least if V is large enough, otherwise the empty list may be optimal, see Theorem 2.4).

Similarly, for a simple parallel system the order $(1, 2, \ldots, n)$ is optimal if and only if $\frac{c_1}{p_1} \leq \frac{c_2}{p_2} \leq \ldots \leq \frac{c_n}{p_n}$. This ordering also defines an optimal elementary policy for MP1 instances with $|M| = 1$ and $B_1 = \emptyset$, with the

caveat that jobs k satisfying

$$c_k > p_k V \tag{3.3}$$

should be removed (Theorem 2.5).

Following Ben-Dov (1981b), the optimal testing procedure for a series-parallel system can be derived using a two-stage approach based on the results above. First, the components are ordered within each parallel subsystem i (the testing equivalent of a module) according to the optimal testing procedure for parallel systems; denote the obtained order by L_i. Subsystem i as a whole then, has an associated probability of failure $\theta_i = \prod_{k \in N_i} q_k$ and an associated expected cost

$$\kappa_i(L_i) = \sum_{k=1}^{|L_i|} (\prod_{l=1}^{k-1} q_{[l]_{L_i}}) c_{[k]_{L_i}},$$

where empty products are taken to be 1. The symbol $[k]_\alpha$ denotes the object in the k-th position according to a given permutation α. When the permutation to be used (in this case L_i) is clear from the context, it will be omitted. The probability of success for subsystem i is denoted by $\pi_i = 1 - \theta_i$.

Second, the subsystems (modules) are ordered according to the optimal testing procedure for series systems, so in non-decreasing order of $\frac{\kappa_i}{\theta_i}$. Denote the obtained order by σ; an optimal order is $(L_{[1]_\sigma}, L_{[2]_\sigma}, \ldots, L_{[m]_\sigma})$. Testing of a subsystem stops as soon as one working component has been found; testing of the system halts as soon as one subsystem has failed or all subsystems are known to be successful. Only in the latter case, the overall system functions.

3.2.3 Reliability importance

An interesting concept from the testing literature is the reliability importance of a component, first defined by Birnbaum (1969) and used in exact

testing algorithms, such as the one by Ben-Dov (1981a), and in approximation testing algorithms, such as by Jędrzejowicz (1983). In Section 3.3.2, we use this concept in the development of a heuristic procedure for MP1.

Let the *reliability function* h for a series-parallel system with system function as in (3.1) be the probability that the system will be working contingent on the vector of success probabilities $\mathbf{p}$, so

$$h(\mathbf{p}) = \mathbb{E}[f(\mathbf{X})] = \prod_{i \in M} \left(1 - \prod_{k \in N_i} q_k \right) = \prod_{i \in M} \pi_i.$$

Following Birnbaum (1969), we define the *reliability importance* I_k of component k as the partial derivative of h with respect to p_k; this is a measure for the component's importance in determining whether the system will be up or down. If I_k is zero then the component is irrelevant. The larger I_k, the more crucial or relevant the component is for the overall system success. Assuming $k \in N_j$, we obtain:

$$\begin{aligned} I_k(\mathbf{p}) \equiv \frac{\partial h}{\partial p_k}(\mathbf{p}) &= -\frac{\partial h}{\partial q_k}(\mathbf{p}) \\ &= \prod_{\substack{i \in M \\ i \neq j}} \left(1 - \prod_{l \in N_i} q_l \right) \prod_{\substack{l \in N_j \\ l \neq k}} q_l = \prod_{\substack{i \in M \\ i \neq j}} \pi_i \prod_{\substack{l \in N_j \\ l \neq k}} q_l. \quad (3.4) \end{aligned}$$

We conclude that a job will be more important when, relatively to the other modules, its module has a low probability of success, and when compared to the other jobs in its module, the job itself has a high probability of success. This is intuitive: since all modules have to succeed, modules with low success probability are more important. Conversely, since within a module the success of one job is sufficient, jobs with high success probability are more important.

3.3 A greedy heuristic

The optimal two-stage procedure for USPST is not directly applicable to MP1 because of two major differences. First, the MP1 setting includes

precedence constraints, so the ordering produced by the USPST algorithm may not be feasible. Second, the list defining an elementary testing policy contains all jobs, while job selection may be beneficial to the MP1 objective. In this section, we propose four different greedy heuristics for MP1. For ease of presentation and understanding, we present the algorithms sequentially from simplest to most complicated: each algorithm is an improvement of the previous one. The described procedures for MP1 are inspired by the exact algorithm for USPST, and we will consider EMS policies only, which can be conveniently represented by their compatible order list. Algorithm Greedy 1 generates an initial list of all jobs. Greedy 2 seeks to improve this initial solution by removing jobs from the list. Greedy 3 handles precedence constraints somewhat more intelligently than Greedy 1. In Greedy 4 we incorporate a randomization step to find better module orderings. For ease of notation define $M' := M \setminus \{0, m+1\} = \{1, \ldots, m\}$ and $A' := A \setminus \{(i,j) \,|\, i = 0 \text{ or } j = m+1\}$.

3.3.1 Greedy 1: An initial list

In this section, we describe a simple heuristic (referred to as Greedy 1) that generates an EMS policy for MP1 starting from the solution for the related USPST problem. In order to satisfy the precedence constraints between jobs inside each module as well as between the modules, a reordering of the jobs and the modules in the USPST solution is necessary. The first stage of the procedure of Ben-Dov (1981b) provides an ordering of the jobs L_{i0} for each module i based on the cost-success probability ratio. Next, we iteratively build a feasible job ordering L_i for each module i as follows. At each step, we greedily select the first unscheduled 'available' job appearing in L_{i0} to be the next job in L_i, where a job is available if it has either no predecessors or if all its predecessors are already in L_i. Next, we can apply the second stage of Ben-Dov's procedure: we compute for each module i a cost κ_i (based on the list L_i) and a failure probability θ_i and compute a module ordering σ_0 based on the ratios κ_i/θ_i. Finally, we apply the same greedy procedure that transforms the module ordering σ_0 into

Algorithm 1 Basic subroutine to produce a linear extension of a poset

Input: poset (Z, E) and permutation L_0 of Z

Output: Algorithm1$((Z, E), L_0) \equiv$ linear extension L of (Z, E)

1: Initialize $L = \emptyset$;
2: Initialize $\mathcal{E} = \{i \in L_0 : j \in L \text{ for all } (j, i) \in E\}$;
3: **while** $L_0 \neq \emptyset$ **do**
4: Let j^* be the first element of L_0 that belongs to $\mathcal{E}$;
5: Append j^* to L and remove it from L_0;
6: Update $\mathcal{E}$;
7: **end while**
8: **return** L

an ordering σ satisfying the module precedence constraints. The greedy subroutine that is used in Greedy 1 to transform a given job or module order into one that satisfies the precedence constraints, is described in detail in Algorithm 1; it transforms a given permutation into a linear extension of a partially ordered set (poset). Notice that in the computation of the set of eligible activities $\mathcal{E}$ (lines 2 and 6 of Algorithm 1), we only need to consider the immediate predecessors. Therefore, we use the transitive reduction[1] of $\mathcal{E}$ in the implementation of Algorithm 1.

To illustrate the functioning of this procedure, we consider the module depicted in Figure 3.1. Assume that in the absence of the precedence constraints, an optimal order within this module is $L_{i0} = (3, 2, 4, 5, 1)$ (ordered in non-decreasing $\frac{c_i}{p_i}$). Greedy 1 then calls Algorithm 1 to transform L_{i0} into $L_i = (2, 4, 5, 1, 3)$.

3.3.2 Greedy 2: Deciding which jobs not to schedule

In order to decide which jobs not to include, we compare the expected incremental benefit of scheduling a job with the expected incremental cost.

[1] To compute the transitive reduction of a poset (V, O) we remove all elements (i, l) of O for which there is an element $j \in V$ such that $(i, j) \in O$ and $(j, l) \in O$.

Greedy 1

Input: MP1 instance

Output: List L defining an EMS policy

1: **for all** $i \in M'$ **do**
2: Sort elements k of N_i in non-decreasing order of $\frac{c_k}{p_k}$, put the result in L_{i0};
3: $L_i \leftarrow \text{Algorithm1}((N_i, B_i), L_{i0})$;
4: Compute $\kappa_i(L_i)$ and θ_i;
5: **end for**
6: Sort elements i of M' in non-decreasing order of ratio $\frac{\kappa_i(L_i)}{\theta_i}$, put the result in σ_0;
7: $\sigma \leftarrow \text{Algorithm1}((M', A'), \sigma_0)$;
8: Set $L \leftarrow (L_{[1]_\sigma}, \ldots, L_{[m]_\sigma})$;
9: Return L;

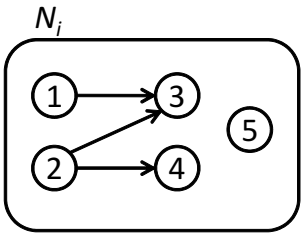


Figure 3.1: *A module with precedence constraints between jobs*

The expected payoff when scheduling all jobs is $EP = h(\mathbf{p})V$. Not scheduling job k amounts to setting its p_k equal to 0. Denote by $\mathbf{p}^{(k)}$ the vector $\mathbf{p}$ with a zero at component k, i.e., $\mathbf{p}^{(k)} = (p_1, \ldots, p_{k-1}, 0, p_{k+1}, \ldots, p_n)$. We derive the following first-order Taylor approximation for $h(\mathbf{p})$ around the point $\mathbf{p}^{(k)}$:

$$h(\mathbf{p}) \approx h(\mathbf{p}^{(k)}) + 0 \cdot \frac{\partial h}{\partial p_1}(\mathbf{p}^{(k)}) + \cdots + p_k \cdot \frac{\partial h}{\partial p_k}(\mathbf{p}^{(k)}) + \cdot + 0 \cdot \frac{\partial h}{\partial p_n}(\mathbf{p}^{(k)}). \quad (3.5)$$

This result can be stated as an equality, since h is linear in each of its arguments, so the first-order approximation of Equation (3.5) actually holds

with equality. By (3.4), we have $\frac{\partial h}{\partial p_k}(\mathbf{p}^{(k)}) = I_k(\mathbf{p}^{(k)}) = I_k(\mathbf{p})$, and so the loss in expected payoff from removing job $k \in N_j$ is given by:

$$\Delta EP = h(\mathbf{p})V - h(\mathbf{p}^{(k)})V = p_k I_k(\mathbf{p})V = \left(\prod_{\substack{i \in M \\ i \neq j}} \pi_i \prod_{\substack{l \in N_j \\ l \neq k}} q_l \right) p_k V.$$

Note that this derivation ignores within-module precedence relationships. To maintain a feasible solution only jobs without successor jobs in that module can be removed.

We denote the expected cost by EC. Contrary to EP, the value of EC depends on the order list at hand. We derive the incremental change compared to a given list $L = (L_{[1]_\sigma}, \ldots, L_{[m]_\sigma})$ of all jobs, as generated by Greedy 1 for instance. Denote by $\mathcal{F}(j)$ and $\mathcal{G}(j)$ the set of modules scheduled (according to a given EMS-policy) before and after a module j, respectively. Note that $\{\mathcal{F}(j), \{j\}, \mathcal{G}(j)\}$ is a partition of M, for any $j \in M$. For a module j, the expected cost reduction obtained by removing the last job k from L_j is:

$$\Delta EC = \left(\prod_{i \in \mathcal{F}(j)} \pi_i \prod_{\substack{l \in N_j \\ l \neq k}} q_l \right) (c_k + p_k \Gamma(j)),$$

with $\Gamma(j)$ the expected cost of the project after module j is completed. Indeed, when all modules before j are successful, and all jobs in N_j have failed, then if we are willing to forfeit the extra attempt offered by job k, not only will we save its own cost c_k, but if job k would have been successful, the expected cost of all modules further on in the schedule will no longer have to be laid out (since job k is removed from the list). The expected cost $\Gamma(j)$ of the project after completion of module j in the EMS-policy can be computed as

$$\Gamma_j = \sum_{i \in \mathcal{G}(j)} \left(\prod_{h \in \mathcal{G}(j) \cap \mathcal{F}(i)} \pi_h \right) \kappa_i. \tag{3.6}$$

Indeed, the expected cost of a module i that is executed after module j only occurs when all the modules scheduled between modules j and i are completed successfully (otherwise the project fails, and the module is never reached).

In conclusion, the net incremental expected profit of the removal of job k that was scheduled last in a module j, is given by:

$$\Delta EC - \Delta EP = \left(\prod_{i \in F(j)} \pi_i \prod_{\substack{l \in N_j \\ l \neq k}} q_l \right) (c_k + p_k \Gamma_j) - \left(\prod_{\substack{i \in M \\ i \neq j}} \pi_i \prod_{\substack{l \in N_j \\ l \neq k}} q_l \right) p_k V \tag{3.7}$$

It seems preferable not to schedule a job when the net incremental expected profit given in (3.7) is positive. In the heuristics that are proposed in the paper, we decide to apply this selection criterium for any job in a module in order to evaluate the desirability of including the job in a schedule, even though the rule is designed only for jobs that are scheduled last in a module. We do this because only removing jobs without successors will not lead to very good solutions if there are very undesirable jobs present that have successor jobs. This rule should be conservative in the sense that we are more likely to keep a job scheduled than in the apparent optimum. Given the precedence constraints that have to be accounted for it is probably better to be conservative. Indeed, when a job is removed from the order list, all its successors need to be removed as well. Since this effect is difficult to factor in explicitly (and we did not in fact incorporate it in the derivations above), for a heuristic rule it seems better to be conservative about removing a job from the list. To derive a conservative heuristic rule we underestimate $\Delta EC - \Delta EP$.

Our negatively biased simplification of (3.7) consists in working with a smaller likelihood of saving the extra costs of the project from module j onwards. To simplify notation, let

$$\prod_{\substack{i \in M \\ i \neq j}} \pi_i = \alpha_j, \quad \prod_{i \in F(j)} \pi_i = \beta_j \text{ and } \prod_{i \in G(j)} \pi_i = \gamma_j,$$

with an empty index set resulting in a product of 1. It holds that $\alpha_j \leq \beta_j$, $\alpha_j = \beta_j \gamma_j$ and

$$\begin{aligned}\Delta EC - \Delta EP &= \beta_j \left(\prod_{\substack{l \in N_j \\ l \neq k}} q_l \right) (c_k + p_k \Gamma_j) - \alpha_j \left(\prod_{\substack{l \in N_j \\ l \neq k}} q_l \right) p_k V \\ &\geq \beta_j \left(\prod_{\substack{l \in N_j \\ l \neq k}} q_l \right) c_k - \alpha_j \left(\prod_{\substack{l \in N_j \\ l \neq k}} q_l \right) p_k (V - \Gamma_j).\end{aligned}$$

We anticipate that it is better not to have job k scheduled at the end of module j if the right-hand side of the latter inequality (which is an underestimation of Equation (3.7)) exceeds zero, so if

$$\beta_j c_k \geq \alpha_j p_k (V - \Gamma_j).$$

With $\alpha_j = \beta_j \gamma_j$, we obtain the following heuristic selection rule:

$$\frac{c_k}{p_k} \geq \gamma_j (V - \Gamma_j). \tag{3.8}$$

That is, remove job $k \in N_j$ from the list if $\frac{c_k}{p_k}$ exceeds a fraction γ_j of the difference between the project payoff V and the expected further project cost Γ_j. Note that for instances with $|M'| = 1$ and $B_1 = \emptyset$, this is exactly the optimal selection rule described by Condition (3.3).

In Greedy 2 we start from the list L returned by Greedy 1, which contains all the jobs. Then, for each module, we remove all jobs that satisfy (3.8) from L together with their successors, resulting in a shorter list L'. Successor jobs must be removed as well to satisfy the precedence constraints. Removing jobs from a module i increases θ_i and decreases κ_i, and thus module i will have a lower ratio κ_i/θ_i. Therefore, we may reorder the modules in L' by applying Algorithm 1 to the updated module order of non-decreasing ratio κ_i/θ_i, resulting in another list L''; this latter list, however, does not necessarily lead to a better policy than the list L'. There is also no guarantee that the selection process will improve the initial list L.

Greedy 2

Input: MP1 instance

Output: List defining an EMS policy

Apply Greedy 1 to obtain a list $L = (L_{[1]}, L_{[2]}, \ldots, L_{[m]})$;
for all $i \in M'$ **do**
  Compute γ_i and Γ_i;
  Let k be the smallest index of L_i such that $\frac{c_{[k]}}{p_{[k]}} \geq \gamma_i(V - \Gamma_i)$;
  if no such index exists **then**
    Set $k = |L_i| + 1$;
  else if $k = 1$ **then**
    Set $k = 2$; {to ensure every module keeps at least one job}
  end if
  Set $L'_i \leftarrow ([1]_{L_i}, [2]_{L_i}, \ldots, [k-1]_{L_i})$;
  Compute $\kappa_i(L'_i)$ and $\theta_i(L'_i) = \prod_{k \in L'_i} q_k$;
end for
Set $L' \leftarrow (L'_{[1]}, \ldots, L'_{[m]})$;
Sort elements $i \in M'$ in non-decreasing order of $\kappa_i(L'_i)/\theta_i(L'_i)$ and put the result in σ'_0;
$\sigma' \leftarrow$ Algorithm1$((M', A'), \sigma'_0)$;
Set $L'' \leftarrow (L'_{[1]_{\sigma'}}, \ldots, L'_{[m]_{\sigma'}})$;
return L, L' or L'' – pick one with the highest expected profit;

Greedy 2 returns from the three lists L, L' and L'', one with the highest expected profit.

Consider again the module depicted in Figure 3.1, for which Greedy 1 found the list $L_i = (2, 4, 5, 1, 3)$. This module is part of a larger test instance, and part of the actual data are reproduced in Table 3.1. Assume that $V = 122, \gamma_i \approx 0.983$ and $\Gamma_i \approx 93.0$ (these values can only be determined on the basis of the entire instance, which is reproduced in Appendix 3.5), then we have a critical value $\gamma_i(V - \Gamma_i) \approx 28.4$. Two ratios exceed this threshold: $\frac{c_1}{p_1}$ and $\frac{c_5}{p_5}$; job $5 = [3]_{L_i}$ is the first job in L_i with a ratio exceeding the threshold. Note that, based on the functioning of Algorithm

Table 3.1: *A module with jobs that are not retained by Greedy 2*

k	c_k	p_k	c_k/p_k
1	46	0.961	47.9
2	10	0.891	11.2
3	2	0.895	2.2
4	12	0.836	14.4
5	41	0.912	45.0

1, the jobs after 5 in L_i either have a more unfavorable ratio (such as 1) or are successors of some job that exceeds the threshold and must therefore be removed in order to obtain a feasible solution (job 3 is a successor of job 1). Consequently, the initial list $L_i = (2, 4, 5, 1, 3)$ is cut off at the job with the lowest ratio exceeding the threshold (job 5), leading to the order $L_i' = (2, 4)$.

3.3.3 Greedy 3: A refinement

On studying the computational results of Greedy 1 and Greedy 2 (a detailed summary of these results is provided in Section 3.4), we observe that modules are frequently not in an optimal order because of the greediness of Algorithm 1. Let us illustrate why Algorithm 1 may fail by considering the following example with three modules, $A = \{(1, 2)\}$ and, for a certain order of their jobs, $\kappa_1 = 4$, $\kappa_2 = 1$, $\kappa_3 = 3$ and $\theta_1 = 0.2$, $\theta_2 = 0.5$, $\theta_3 = 0.2$. The ratios κ/θ are 20, 2 and 15, and the desired order $\sigma_0 = (2, 3, 1)$. Calling Algorithm1((M, A),σ_0) yields $\sigma = (3, 1, 2)$, with expected cost $\kappa_3 + \pi_3(\kappa_1 + \pi_1\kappa_2) = 6.84$. Even though module 2 is clearly preferable and has a low cost, it cannot be placed first because module 1 is a predecessor. Algorithm 1 places module 3 first because module 1 has a ratio that is slightly higher than that of module 3. It is an optimal decision, however, to select module 1 first in order to enable module 2 to come earlier. The

Algorithm 2 Enhanced subroutine to produce a linear extension of a poset

Input: poset (Z, E) and permutation L_0 of Z

Output: linear extension L of (Z, E)

Initialize $L = \emptyset$;
Let $x \leftarrow [1]_{L_0}$;
Let $P(x) \leftarrow \{i \in Z : (i, x) \in E\}$; {predecessors of x}
Let $PP(x) \leftarrow \{i \in Z : (i, j) \in E \text{ for some } j \in P(x)\}$;
if $|P(x)| \leq 2$ and $PP(x) = \emptyset$ **then**
    $L \leftarrow P(x)$ in order of L_0;
    $L_0 \leftarrow L_0 \setminus P(x)$;
end if
while $L_0 \neq \emptyset$ **do**
    Let j be the smallest index of L_0 such that $i \in L$ for all $(i, [j]_{L_0}) \in E$;
    Append $[j]_{L_0}$ to L and remove it from L_0;
end while
return L

order $(1, 2, 3)$ has (lower) expected cost $\kappa_1 + \pi_1(\kappa_2 + \pi_2\kappa_3) = 6$.

The described effect will be significant if the module with the lowest ratio, say module j, (a) has a ratio that is considerably lower than all the other modules, (b) has very few predecessors, (c) these predecessors have a ratio that is not too high relative to the modules other than j, (d) these predecessors do not have any predecessors themselves. In the example, (a) clearly holds since $2 << 15$ and $2 << 20$, (b) holds since module 2 has only one predecessor, (c) holds since 20 does not exceed 15 by orders of magnitude, and (d) holds since module 1 has no predecessors. In an improved version of Algorithm 1, we check whether module j has at most two predecessors (condition (b)) and whether these predecessors have no predecessors themselves (condition (d)). If so, we consider a new schedule in which these predecessors appear first. The enhanced subroutine is described in Algorithm 2. Conditions (a) and (c) are difficult to specify,

Greedy 3

Input: MP1 instance

Output: List defining an EMS policy

1: Let L be the output of Greedy 2;
2: Let $\bar{L}$ be the output of Greedy 2 with subroutine Algorithm 1 replaced by Algorithm 2 when line 7 of Greedy 1 is called, and when line 15 of Greedy 2 is called;
3: **return** L or $\bar{L}$ – pick one with the highest expected profit;

and so we execute both Algorithm 1 and Algorithm 2, and see which one performs best.

The enhanced subroutine can now be implemented in Greedy 3. It will only be used to order modules, not jobs. From the tests on the dataset, we observe that the optimal order of jobs within a module is nearly always obtained, and if not then the loss in quality is usually not very important. We conclude by noting that a problem would arise if we were to apply this subroutine also within modules: a choice would need to be made to which modules to apply Algorithm 2 and to which Algorithm 1, either heuristically, or by checking all possibilities – which would make the runtime of the heuristic exponential in the number of modules.

3.3.4 Greedy 4: Randomizing the module order

The quality of the heuristic selection rule derived for Greedy 2 highly depends on the ordering of the modules and only works well when the module order is optimal or near-optimal. In the previous subsection we have illustrated that the simple greediness of Algorithm 1 may result in module orderings that are far from optimal. For this reason, Greedy 3 orders the modules in a slightly more enhanced manner; improvements, however, are realized only in very specific situations. To overcome this, we have introduced a randomization step in order to increase the likelihood of finding good module orderings. Instead of always choosing from the set $\mathcal{E}$ of

Algorithm 3 RBRS variant of Algorithm 1

Input: poset (Z, E) and permutation L_0 of Z

Output: Algorithm3$((Z, E), L_0) \equiv$ linear extension L of (Z, E)

1: Initialize $L = \emptyset$;
2: Initialize $\mathcal{E} = \{i \in L_0 \,:\, j \in L \text{ for all } (j, i) \in E\}$;
3: **while** $L_0 \neq \emptyset$ **do**
4: Randomly pick one element $j^* \in \mathcal{E}$, where each $j \in \mathcal{E}$ has probability P_j of being selected according to Equation 3.9;
5: Append j^* to L and remove it from L_0;
6: Update $\mathcal{E}$;
7: **end while**
8: **return** L

eligible modules one with lowest ratio κ/θ (see Algorithm 1), we now create the possibility of selecting a module with a higher ratio (Algorithm 3) by means of regret-based random sampling (RBRS) (Drexl, 1991). This idea is also similar to the notion of stochastic ranking that was popularized by evolutionary algorithms (Runarsson and Yao, 2000). The probability P_i of selection of a module i out of the set $\mathcal{E}$ is determined as follows:

$$P_i = \frac{(\rho_i + 1)^\alpha}{\sum_{j \in \mathcal{E}} (\rho_j + 1)^\alpha}, \tag{3.9}$$

with $\rho_i = [\arg\max_{j \in \mathcal{E}} \kappa_j/\theta_j]_{L_0} - [i]_{L_0}$ and where $[j]_{L_0}$ denotes the position of module j in the initial module ordering L_0 (non-decreasing κ/θ). In Greedy 4, we start from the result of Greedy 3 to guarantee producing a solution that is at least as good. Subsequently, we invoke Greedy 2 for different module orderings obtained via Algorithm 3 until a stopping criterion is met. Note that the module order obtained by Algorithm 2 is most likely to be generated by Algorithm 3 as well, so in practice we could probably start also from the solution obtained by Greedy 1.

We choose the values ρ_i of module i in terms of the position of module i in the module ordering L_0 rather than the ratio κ/θ itself, because the

Greedy 4

Input: MP1 instance

Output: List defining an EMS policy

1: Let L be the output of Greedy 3;
2: **while** Stop criterion not met **do**
3: Let $\bar{L}$ be the output of Greedy 2 with subroutine Algorithm 1 replaced by Algorithm 3 when line 7 of Greedy 1 is called;
4: Update L if $\bar{L}$ has higher expected profit;
5: **end while**
6: **return** L;

ratio can take rather extreme values (especially for large modules). The parameter $\alpha \in [0, \infty)$ controls the diversification of the sample of module orderings selected from the population of all possible module orderings. The diversification in the module lists decreases with α. The boundary case of $\alpha = 0$ corresponds to a completely random selection from $\mathcal{E}$ with equal probability $1/|\mathcal{E}|$, whereas in the other extreme of $\alpha = +\infty$, we always select that element of $\mathcal{E}$ appearing first in the initial ordering L_0 (as in Algorithm 1). In Section 3.4.1 we elaborate on the choice of the parameter α and on the stop criterion.

3.4 Computational experiments

In this section, we assess the performance of the algorithms on the same dataset that was used to test the exact algorithms in the previous chapter (only the general MP1 instances with possibly more than one job per module, see Section 2.5.1).

All experiments were run on a Dell Latitude D830 with a 2.5 GHz processor and 3 GB of RAM, running 32 bit Windows Vista. The algorithms were implemented in C++ using Microsoft Visual Studio 2010. We evaluate the performance of the greedy heuristics against the two exact algorithms de-

veloped in Chapter 2: a branch-and-bound (B&B) algorithm that finds an optimal EMS policy (Section 2.4.2), and a dynamic-programming (DP) algorithm that outputs a globally optimal policy (Section 2.4.1). To facilitate comparison, we define the relative optimality gap ROG of an algorithm $\mathcal{A}$ for an instance of MP1 as follows:

$$ROG = \frac{z^* - z(\mathcal{A})}{z^*} \text{ if } z^* \neq 0, \text{ and 0 otherwise,} \tag{3.10}$$

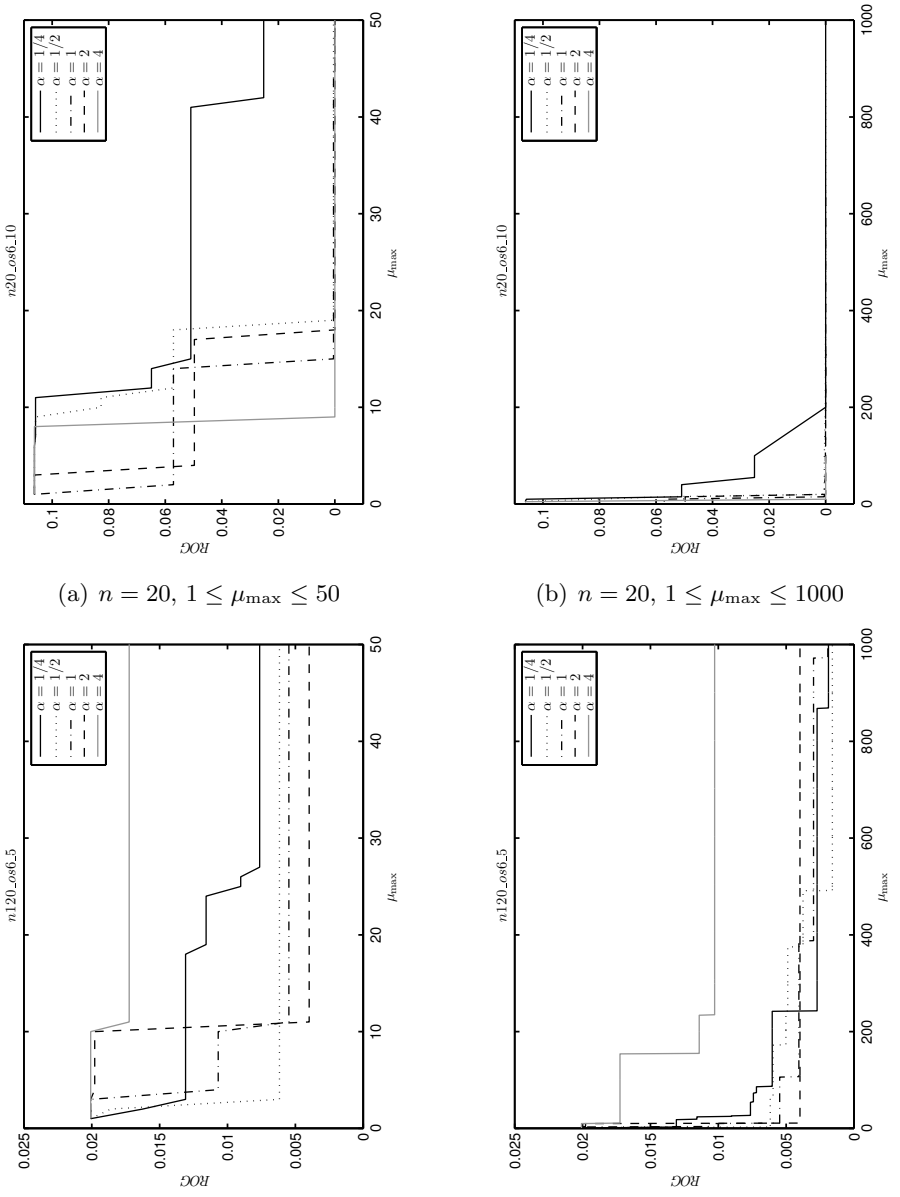
where z^* is the objective value of a globally optimal solution (found by DP) and $z(\mathcal{A})$ the objective of the output of $\mathcal{A}$. Notice that this definition is different from the relative optimality gap $\gamma(\mathcal{C})$ of a policy class $\mathcal{C}$ (see Section 2.2).

Below, we include a discussion of some implementation choices for heuristic Greedy 4 (Section 3.4.1), followed by a presentation of the computational results (Section 3.4.2).

3.4.1 Implementation choices for Greedy 4

For the implementation of algorithm Greedy 4, we need to decide on the stop criterion and on the value of the parameter α. A natural stop criterion is to reach a maximum number $\mu_{\max}$ of different module orders generated by Algorithm 3, is a natural stop criterion. In Figure 3.2 we show the relative optimality gap as a function of $\mu_{\max}$ for different values of α, for a small instance with 20 jobs (Figures 3.2(a) and 3.2(b)), and for a large instance with 120 jobs (Figures 3.2(c) and 3.2(d)). For each instance the left plot focuses only on small values of $\mu_{\max}$ (at most 50), whereas the right plot shows the performance of Greedy 4 when more module orderings are generated (at most 1000). In the following paragraphs, we discuss the conclusions drawn from this figure with respect to a good choice for the parameter α depending on the size of the instance and the choice of the stop criterion.

The performance of Greedy 4 on the dataset was assessed with two different stop criteria. First, when it is desirable that the computation time for



(a) $n = 20$, $1 \leq \mu_{\max} \leq 50$

(b) $n = 20$, $1 \leq \mu_{\max} \leq 1000$

(c) $n = 120$, $1 \leq \mu_{\max} \leq 50$

(d) $n = 120$, $1 \leq \mu_{\max} \leq 1000$

Figure 3.2: *Relative optimality gap as a function of* $\mu_{\max}$ *for different* $\alpha \in \{1/4, 1/2, 1, 2, 4\}$ *of a small instance with 20 jobs (*n20_*os*6_10*) and a large instance with 120 jobs (*n120_*os*6_5*). The left figures only show* $\mu_{\max} \leq 50$*, the right figures have* $\mu_{\max} \leq 1000$*.*

Greedy 4 be of the same order of magnitude as Greedy 1, Greedy 2, and Greedy 3 (i.e., only a fraction of a second), we set $\mu_{\max} = 50$. We will refer to this variant as Greedy 4a. When finding a higher-quality solution is important but still aiming to develop a *fast* heuristic, a time limit of one second is imposed as the second stop criterion; this variant of the algorithm is named Greedy 4b. We can see from Figure 3.2 that a convenient choice for the parameter α depends on the stop criterion selected. In general, we may conclude that a good choice of the parameter α depends on the value of $\mu_{\max}$ and on the size of the instance.

In Greedy 4a, only a small number of (different) module orderings are generated. Typically, the total number of possible module orderings is far higher than $\mu_{\max} = 50$, and good orderings need to be found with only a few attempts (see Figure 3.2(c)). It is to be expected that orderings that do not differ too much from the ordering generated by Algorithm 1 are likely to be of reasonable quality. Figure 3.2(c) shows that too much diversification ($\alpha = 1/4, 1/2$) will lead to worse solutions compared to a lower diversification level ($\alpha = 1, 2$). It is also crucial, however, to allow for a certain level of flexibility in the orderings, as illustrated by the poor performance of $\alpha = 4$. From Figure 3.2(a) we observe that the performance of Greedy 4 is less sensitive to the level of diversification when the instance is small. This is to be expected: the total number of module orderings is also far lower. In general we infer that $\alpha = 2$ is a good choice for Greedy 4a.

Greedy 4b works with a time limit of one second; within this time we can produce close to 2000 different module orders for the large instances. Since more lists can be generated, we expect a higher diversification level to perform better; Figure 3.2(d) suggests $\alpha = 1/2$ as the best choice. It is to be noted that for the small instance of 20 jobs we find only 1000 different orders for $\alpha = 1/4$ (within a time limit of one second). Higher values of α result in a high number of duplicate orderings: when $\alpha = 4$, for example, we encounter only 97 different module lists within one second.

3.4.2 Computational results

In Table 3.2, the average relative optimality gaps are given for the general MP1 instances with $n \in \{10, 20, 30, 40\}$. The B&B cannot solve all the corresponding 120 instances to guaranteed optimality within a time limit of 30 minutes; especially larger instances and instances with a lower OS are more difficult to solve. We therefore report the average performance only for the 85 instances solved by the B&B. The average CPU time is less than 0.01 seconds for the DP algorithm and for the three deterministic greedy algorithms. The average CPU time for Greedy 4a is 0.66 seconds: some time is needed for generating $\mu_{\max}$ different module orders (or for reaching a time limit of one second, whichever comes first) – this will not be the case for larger instances (see next paragraph of this section). For the 85 instances solved by the B&B, the average ROG is about 0.01%, so the expected profit is virtually the same in each case. The ROG of our final heuristic Greedy 4b is 0.13%. When only 50 module orderings are generated, the gap increases by 0.05% up to 0.18%. The variation coefficient (standard deviation divided by average) of the average ROG of the randomized heuristics Greedy 4a and Greedy 4b in Table 3.2 is 0.11 and 0.015, respectively. The 1.97% gap of the best deterministic heuristic (Greedy 3) is significantly higher than the gap of Greedy 4. Greedy 2, with a simpler ordering subroutine, achieves 3.21% and the average gap of Greedy 1 is yet slightly higher at 4.76%. This means that the average extra profit achieved by making a selection of jobs (not scheduling all jobs) amounts to 1.55% of the global optimum. Looking into the dataset in more depth, we observe that in 13 instances the optimal EMS policy generated by B&B does not schedule all available jobs. In seven out of these, Greedy 3 makes the same selection. In three instances, Greedy 3 makes a selection that is somewhat larger than the optimal one. For three other instances, Greedy 3 fails to make a selection and schedules all available jobs, while B&B finds an optimal policy that only schedules a subset of N. Greedy 4b finds an optimal EMS policy for all but one of these 13 instances.

Table 3.2: *Average performance over 85 instances with* $n \in \{10, 20, 30, 40\}$

Algorithm	Resulting policy	CPU (s)	*ROG*
DP	Global optimum	< 0.01	0.00%
B&B	Optimal EMS	81.28	0.01%
Greedy 4b	Randomized heuristic EMS	1.00	0.13%
Greedy 4a	Randomized heuristic EMS	0.66	0.18%
Greedy 3	Deterministic heuristic EMS	< 0.01	1.97%
Greedy 2	Deterministic heuristic EMS	< 0.01	3.21%
Greedy 1	Deterministic heuristic EMS	< 0.01	4.76%

For the instances with $n > 40$, the results of the heuristic can only be compared to the DP, which can solve 171 out of the 240 remaining instances without memory overrun; the results for these 171 instances are summarized in Table 3.3. The unsolved instances are again those with high n and low OS. The average runtime of the DP increases significantly when the instances become larger: it now needs 83.07 seconds on average; especially instances with a low order strength need more time. The runtime of each greedy algorithm is less than 0.01 seconds, except for Greedy 4b (which has an imposed time limit). Compared to Table 3.2, we observe that Greedy 4a is significantly faster for these larger instances: finding 50 different orderings turns out to be far easier for higher n. The gaps for Greedy 1, 2 and 3 are slightly lower than for the instances solved by B&B (Table 3.2). A possible explanation might be that the optimal objective value increases with n (which is inherent in the way the instances were created). The improvement of Greedy 2 vis-à-vis Greedy 1 is substantially smaller, at a mere 0.18%. Since optimal EMS policies are not available for these instances, we cannot assess whether this is because our heuristic selection rule (Equation 3.8) performs worse for larger instances, or because the instances in this part of the dataset simply did not require a selection of jobs. The improvement realized by Greedy 4 is smaller for small instances

Table 3.3: *Average performance over 171 instances with* $n \in \{50, 60, \ldots, 120\}$

Algorithm	Resulting policy	CPU (s)	*ROG*
DP	Global optimum	83.07	0.00%
Greedy 4b	Randomized heuristic EMS	1.00	0.50%
Greedy 4a	Randomized heuristic EMS	< 0.01	0.94%
Greedy 3	Deterministic heuristic EMS	< 0.01	1.64%
Greedy 2	Deterministic heuristic EMS	< 0.01	1.67%
Greedy 1	Deterministic heuristic EMS	< 0.01	1.85%

(Table 3.2) than for larger instances (Table 3.3), but the average relative optimality gap for our best heuristic, Greedy 4b, is still small (only a half percent).

For the instances that were not solved by the DP, we cannot determine *ROG* because a global optimum is not known. In this case, we can only evaluate the relative gap of our algorithms with respect to the best solution found, i.e. the expected profit found by Greedy 4b. The first column of Table 3.4 contains the average gaps for the heuristics compared to Greedy 4b over the 69 instances that were not solved by DP. We observe an average improvement with respect to Greedy 1 and Greedy 2 of 0.52%. The selection rule seems to fail in Greedy 2 (no improvement by selection in any of these 69 instances). The improvement of Greedy 3 and Greedy 4 is realized by a reordering of the modules. Here also, no selection of jobs occurs; again we cannot evaluate whether this is because the selection rule of Equation 3.8 becomes of lesser quality for higher values of n, or rather because no selection is required. The second column in Table 3.4 reports the averages over all 360 instances of the dataset.

Figure 3.3 depicts the *ROG* of Greedy 4b as a function of n, for each of the three values of *OS*. The curves apply to the 289 instances solved by DP; each observation is the average for the solved instances of the setting

Table 3.4: *Average relative improvement of Greedy 4b over the remaining 69 instances that were not solved by DP, and over all 360 instances of the dataset*

Algorithm	69 unsolved instances	All 360 instances
Greedy 4b	0.00%	0.00%
Greedy 4a	0.26%	0.38%
Greedy 3	0.46%	1.32%
Greedy 2	0.52%	1.72%
Greedy 1	0.52%	2.18%

considered (at most 10). No clear patterns arise; the significant fluctuations are presumably due simply to random idiosyncrasies in the dataset.

3.5 Conclusions

In this chapter, we continue our study of the problem of modular project scheduling on one machine (MP1), for which exact scheduling algorithms have been developed in Chapter 2. When only few precedence constraints are imposed, however, these exact algorithms either run out of memory or require increasing amounts of time when the number of activities increases. The goal of this chapter was to develop a heuristic that produces 'good' schedules for such projects, i.e., schedules with a high expected profit, while requiring only very limited CPU time and computer memory.

MP1 is closely related to the series-parallel sequential testing problem. The optimal two-step procedure for the testing problem without precedence constraints is the starting point for the development of a fast heuristic procedure for MP1, in which we produce an order list that defines an elementary module-sequence scheduling policy. Four variants are proposed, in increasing order of complexity. The starting point is a simple greedy algorithm, and this is stepwise refined and complemented with a random-

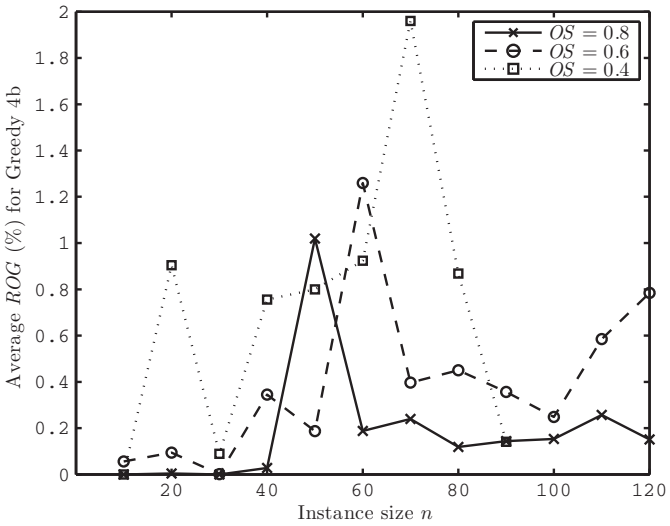


Figure 3.3: *Average relative optimality gap of Greedy 4b*

ization step in the final variant. Based on computational experiments on a large dataset, we find that the algorithms output near-optimal in negligible runtimes: the average optimality gap for instances for which an optimal solution is known, is quite limited (0.5% or less). We produce approximate solutions for complex instances that have not been solved by an exact algorithm in the earlier reference, and this within runtimes of at most one second.

A possible avenue for further research on the topic of modular project scheduling is the exploration of the multi-mode character of the modules: every module can also be seen as a 'composite' activity, which can be executed in multiple 'modes', each mode corresponding to one possible selection of activities within the module. The literature on (especially project) scheduling has already looked into various multi-mode problems including time-cost trade-offs (De et al., 1995) and time-resource trade-offs (De Reyck and Demeulemeester, 1998). The standard multi-mode problem is a generalization that, in addition to the time-cost and time-resource trade-offs, also covers resource-resource trade-offs and the use of multiple

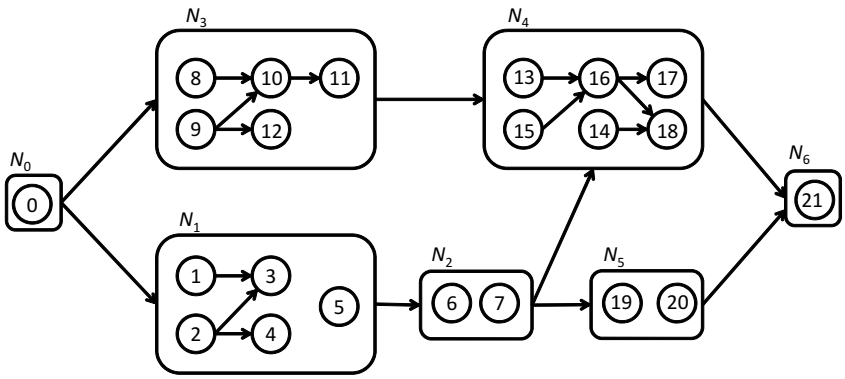


Figure 3.4: *Precedence network*

types of renewable and nonrenewable resources (Talbot, 1982). To the best of our knowledge, multiples modes corresponding to multiple selections of (sub-)activities contained in the original activity (in our case: the module) have not yet been described in earlier work. From an algorithmic viewpoint, an obvious further step towards finding high-quality solutions to MP1 would be the development of a meta-heuristic procedure (for instance, tabu search or genetic algorithms, or any other meta-heuristic framework); in light of the low optimality gaps that are already achieved by the (faster) algorithms proposed in this paper, however, we have not pursued this option. For the same reason, the option of approximate dynamic programming for finding higher-quality heuristic solutions has also not been examined.

Appendix: an instance with $n = 20$

In this appendix, for illustration purposes, we describe the outputs of the algorithms proposed in this text for the instance depicted in Figure 3.4. The numerical data for this instance can be found in Table 3.5. The instance is part of the dataset (instance name $g\_n20\_os6\_4$). The initial

ordering generated by Greedy 1 is

$$L = (2, 4, 5, 1, 3; 6, 7; 19, 20; 8, 9, 10, 11, 12; 13, 15, 16, 17, 14, 18),$$

which is easy to verify manually with the data from Table 3.5 and the pseudocode of Greedy 1. Greedy 2 removes jobs 1, 3 and 5 and the order of the modules is redetermined. In this case, the heuristic order of the modules does not change ($L' = L''$ in lines 13-16 of the pseudocode of Greedy 2), and

$$L' = (2, 4; 6, 7; 19, 20; 8, 9, 10, 11, 12; 13, 15, 16, 17, 14, 18).$$

The expected profit of L and L' is approximately 14.72 and 15.05, respectively, so Greedy 2 will return L'. Greedy 3 finds the same solution. An optimal EMS policy found via B&B turns out to be slightly better, with an expected profit of 15.32. An optimal ordering is given by

$$(8; 2, 4, 5, 1, 3; 6, 7; 19, 20; 13, 15, 16, 17, 14, 18);$$

this list is also found by the two implementations of Greedy 4.

Table 3.5: *Costs and probabilities*

k	c_k	p_k	c_k/p_k
1	46	0.961	47.9
2	10	0.891	11.2
3	2	0.895	2.2
4	12	0.836	14.4
5	41	0.912	45.0
6	32	0.977	32.8
7	33	0.844	39.1
8	15	0.833	18.0
9	41	0.922	44.5
10	16	0.978	16.4
11	15	0.972	15.4
12	24	0.903	26.6
13	17	0.856	19.9
14	46	0.825	55.8
15	22	0.860	25.6
16	33	0.966	34.2
17	45	0.902	49.9
18	42	0.906	46.4
19	14	0.898	15.6
20	41	0.866	47.3
V	122		

Chapter 4

Discrete sequential search with group activities

In this chapter, we investigate the correctness of a conjecture appearing in Wagner and Davis (2001), where an integer-programming model is presented for the single-item discrete sequential search problem with conjunctive and disjunctive group activities. Based on their experiments, they conjecture that the special case with only conjunctive group activities can be solved as a linear program. First, we provide a counterexample for which the optimal value of the linear program they propose is different from the optimal value of the integer-programming model, hence contradicting their conjecture for the specific linear program that they specify. Next, we prove that the conjunctive case is strongly NP-hard, which implies that, unless P = NP, it is impossible that there exists any compact linear program for solving this problem. Furthermore, we show that the disjunctive case is also strongly NP-hard, and we discuss some special cases that can be solved in polynomial time.

This chapter is based on joint work with dr. F. Talla Nobibon and prof. dr. R. Leus. Its content is based on a published paper in Decision Sciences (Talla Nobibon et al., 2013) and Research Report KBI_1313 (Coolen et al., 2013).

4.1 Introduction

The *discrete sequential search problem with group activities* as defined by Wagner and Davis (2001), is as follows. A single object is hidden in one of n boxes and the probability that box k contains the object is π_k ($\sum_{k=1}^{n} \pi_k = 1$). The boxes are searched one at a time and the cost of searching box k is t_k. If the box containing the object is searched then the object is detected with certainty. When the first $n-1$ searches are negative, it is certain that the item is hidden in the final unsearched box. It is assumed that this final box must still be searched (and therefore its cost is incurred). There are also m 'group activities'. Each group activity ℓ has a cost R_ℓ and is associated with a subset $S_\ell \subseteq \{1, \ldots, n\}$ of boxes. Note that some boxes may appear in more than one subset S_ℓ. The group activities are said to be *conjunctive* if any box can be searched only when all the group activities in which it appears have been performed whereas for *disjunctive* group activities, a box can be searched as soon as at least one of the group activities in which it appears has been executed. The goal is to find a sequence (defining a search strategy) in which the boxes are to be searched and the group activities are to be performed so as to minimize the expected cost while satisfying the precedence constraints imposed by the group activities. We refer to the discrete sequential search problem with exclusively conjunctive, respectively disjunctive, group activities as Problem 1, respectively Problem 2.

Discrete sequential search problems have applications in various areas such as quality control (Mitten, 1960), research and development (Joyce, 1971), diagnostic tests on components of complex radar, missile, and communications systems (Gluss, 1959). In the diagnostic sequencing problem, illustrations of a group activity include removing an access cover, draining fluids, disconnecting a power supply, etc., which must occur before a set of components can be tested. In the problem of sequencing tasks in a research-and-development project, a group activity may represent a facility that must be constructed before a set of tasks can be completed (Wagner

and Davis, 2001).

In 2001, Wagner and Davis (Wagner and Davis, 2001) presented an integer-programming model for the discrete sequential search problem including both conjunctive and disjunctive group activities. Based on their experiments, they conjectured that the conjunctive case (Problem 1) can be solved as a linear programming problem.

In Section 4.2, we describe a counterexample for which the optimal value of the linear program proposed by Wagner and Davis is different from the optimal value of the integer-programming model, hence contradicting the conjecture. In Section 4.3, we show that Problem 1, respectively Problem 2, is equivalent to scheduling a set of jobs on a single machine to minimize the total weighted completion time with *and*, respectively *or*, precedence constraints, represented by a special bipartite graph. We exploit this equivalence to establish NP-hardness results for Problem 1 in Section 4.4, and for Problem 2 in Section 4.5. We further study some polynomially solvable cases of Problem 1 and Problem 2 in Section 4.6. Finally, a conclusion and an outlook for further work is given in Section 4.7.

4.2 Counterexample

Wagner and Davis (2001) propose an integer-programming model for the single-item discrete sequential search problem with conjunctive group activities using the binary decision variable δ_{ij} that takes the value 1 if and only if box i is searched before box j for $i \neq j$, and $\delta_{ii} = 1$ for $i = 1, 2, \ldots, n$. They also use the binary variable $T_{\ell j}$ that takes the value 1 if group activity ℓ is performed before before box j is searched, and 0 otherwise. The integer-programming model (for $n \geq 3$) is then defined as follows:

Table 4.1: *Box properties (left) and group-activity properties (right) of the counterexample.*

box i	1	2	3	gr. act. ℓ	1	2	3
t_i	14	13	16	R_ℓ	7	10	6
π_i	7/19	5/19	7/19	S_ℓ	$\{1,2\}$	$\{1,3\}$	$\{2,3\}$

$$\min \quad \sum_{j=1}^{n} \pi_j \left(\sum_{i=1}^{n} t_i \delta_{ij} + \sum_{\ell=1}^{m} R_\ell T_{\ell j} \right) \tag{4.1}$$

s.t.

$$\delta_{ij} + \delta_{ji} = 1, \qquad 1 \le i < j \le n, \tag{4.2}$$

$$\delta_{ij} + \delta_{jk} + \delta_{ki} \le 2, \quad i,j,k = 1,\ldots,n \ (i \ne j; i,j \ne k), \tag{4.3}$$

$$\delta_{ii} = 1, \qquad i = 1,\ldots,n, \tag{4.4}$$

$$T_{\ell j} \ge \delta_{ij}, \qquad j = 1,\ldots,n;\ \ell = 1,\ldots,m;\ i \in S_\ell, \tag{4.5}$$

$$\delta_{ij} \in \{0,1\}, \qquad i,j = 1,\ldots,n, \tag{4.6}$$

$$T_{\ell j} \in \{0,1\}, \qquad j = 1,\ldots,n;\ \ell = 1,\ldots,m. \tag{4.7}$$

The objective function (in Equation (4.1)) minimizes the expected total cost expressed as the sum of two parts. Indeed, if the object is hidden in box j (which happens with probability π_j), then the total cost of retrieving the object from box j equals the sum of the total cost of previously searched boxes (including box j) and the total cost of previously executed group activities. The constraints of Equation (4.2) stipulate that two different boxes cannot be searched at the same time. The set of constraints in Equation (4.3) enforces the transitivity property for three boxes. The set of constraints in Equation (4.4) enforces that $\delta_{ii} = 1$ for $i = 1, 2, \ldots, n$. The set of constraints in Equation (4.5) ensures that if a group activity is performed before searching a given box then the cost of that group activity is included in the expected cost of searching that box, and finally the last sets of constraints shown in Equations (4.6) and (4.7) represent

Table 4.2: *Values of δ_{ij} (left) and $T_{\ell j}$ (right) for an optimal solution to the counterexample.*

	box j		
box i	1	2	3
1	1	1	1
2	0	1	0
3	0	1	1

	box j		
gr. act. ℓ	1	2	3
1	1	1	1
2	1	1	1
3	0	1	1

integrality constraints. Wagner and Davis relax the integrality constraints in Equations (4.6) and (4.7) into:

$$0 \leq \delta_{ij} \leq 1, \qquad i, j = 1, 2, \ldots, n, \tag{4.8}$$

$$0 \leq T_{\ell j} \leq 1, \qquad j = 1, 2, \ldots, n;\ \ell = 1, \ldots, m, \tag{4.9}$$

and they use the model shown in Equations (4.1)–(4.5), and (4.8)–(4.9) (which is a linear program) to solve a number of randomly generated instances of the single-item discrete sequential search problem with conjunctive group activities. For all instances tested, the linear program had an integer optimal solution, meaning a solution with $\delta_{ij} \in \{0, 1\}$ and $T_{\ell j} \in \{0, 1\}$. Based on these results, they conjecture that the single-item discrete sequential search problem with conjunctive group activities can be solved as a linear program.

We now provide an example with three boxes and three group activities for which the optimal objective value of the linear program is different from the optimal value of the integer-programming model, hence contradicting their conjecture. Consider the instance whose properties are described in Table 4.1. An optimal solution to the integer-programming model (4.1)–(4.7) applied to this instance, has the objective value of 1836/38 and is displayed in Table 4.2. The linear program (4.1)–(4.5), (4.8), (4.9), yields the optimal objective value of 1825/38 and an optimal solution is depicted in Table 4.3. A LINGO implementation of the counterexample is provided

Table 4.3: *Values of δ_{ij} (left) and $T_{\ell j}$ (right) for an optimal solution to the linear program for the counterexample.*

	box j		
box i	1	2	3
1	1	0.5	0.5
2	0.5	1	0.5
3	0.5	0.5	1

	box j		
gr. act. ℓ	1	2	3
1	1	1	0.5
2	1	0.5	1
3	0.5	1	1

in Figure 4.1. This counterexample clearly contradicts the conjecture of Wagner and Davis (2001) stated as follows (p. 570): "Thus, we make the conjecture that the problem can be solved as a linear program (with the exception of $\lambda_{m_1\, m_2}$), knowing that this property is not a result of the total unimodularity of A." The mentioned variables $\lambda_{m_1\, m_2}$ do not appear in our model because we focus only on the case with conjunctive group activities. The disjunctive case will be treated in Section 4.5. For a practical decision maker, the fact that the variables δ_{ij} may not all be integers constitutes a problem, since these variables correctly represent sequencing decisions only when they take binary values.

4.3 Link with the scheduling literature

Consider the problem where each of n jobs is to be processed without interruption on a single machine that can handle only one job at a time. Job i ($i = 1, \ldots, n$) becomes available at time zero (no release dates), requires a processing time p_i and has a non-negative weight w_i. The objective is to find a processing order of the jobs that minimizes the sum of weighted completion times $\sum_{i=1}^{n} w_i C_i$, where C_i is the time at which job i completes in the given schedule. In standard notation (Graham et al., 1979) the problem is referred to as $1||\sum w_i C_i$. This generic problem has an $O(n \log n)$ algorithm based on Smith's rule (Smith, 1956), which schedules jobs in

```
MODEL:

SETS:
  BOXES / 1..3 /: c,p;
  GROUPACT / 1..3/: R;
  PAIR1(BOXES,BOXES): delta;
  PAIR2(GROUPACT,BOXES): S, T;
ENDSETS

DATA:
  c = 14 13  16;
  p = 7 5  7;
  R = 7 10 6;
  S = 1 1  0
      1 0  1
      0 1  1;
ENDDATA

  !objective function: minimize total expected cost;
  MIN=@SUM(BOXES(j): p(j)*@SUM(BOXES(i): c(i)*delta(i,j)))+
      @SUM(GROUPACT(l):@SUM(BOXES(j): p(j)*T(l,j))));

  !two different boxes cannot be searched at the same time;
  @FOR(BOXES(i): @FOR(BOXES(j)| i#LT#j: delta(i,j)+delta(j,i)=1));

  !transitivity property for the three boxes;
  @FOR(BOXES(i):@FOR(BOXES(j): @FOR(BOXES(k)|(i#NE#j)#AND#(j#NE#k):
       delta(i,j)+delta(j,k)+delta(k,i)<=2)));

  !delta(i,i)=1;
  @FOR(BOXES(i): delta(i,i)=1);

  !include cost group activity in cost of a box when preceding;
  @FOR(BOXES(j): @FOR(GROUPACT(l): @FOR(BOXES(i):
       T(l,j) >= S(l,i)*R(l)*delta(i,j))));

  !integrality constraints;
  @FOR(PAIR1: @BIN(delta)); !comment out this line for LP relaxation;
END
```

Figure 4.1: LINGO *implementation of the counterexample*

such a way that for all pairs of jobs i and j, job i is executed before job j if $p_i w_j < p_j w_i$.

The discrete sequential search problem without group activities is equivalent to $1||\sum w_i C_i$ in the sense that any algorithm for the first problem can be used to solve the second problem, and vice versa. On the one hand, when we are given an instance of the search problem, we can construct an instance of $1||\sum w_i C_i$ if we associate with each box i a job i with weight $w_i := \pi_i$ and processing time $p_i := t_i$. In this way the total expected cost of finding the hidden object for any search order of the boxes equals the total weighted completion time of the processing order of the corresponding jobs. On the other hand, for any instance of $1||\sum w_i C_i$, we create for each job i a box i with $\pi_i := w_i/W$ and $t_i := p_i$, where $W = \sum_{i=1}^{n} w_i$. If z is the total expected search cost of some order of the boxes, then the total weighted completion time of the corresponding order of jobs is Wz. By this equivalence, it follows that the discrete sequential search problem without group activities can be solved in time $O(n \log n)$ by ordering the boxes k in non-decreasing order of t_k/π_k.

The problem $1||\sum w_i C_i$ has been extended following several directions, including the addition of precedence constraints among jobs. When precedence constraints are included, the problem is written as $1|prec|\sum w_i C_i$. In the literature, precedence constraints are specified by a directed acyclic graph $G = (J, A)$, where $J = \{1, \ldots, n\}$ and an arc $(i, j) \in A$ indicates that job i must be executed before job j. Lenstra and Rinnooy Kan (1978) show that $1|prec|\sum w_i C_i$ is strongly NP-hard when G is an arbitrary directed acyclic graph, even if each job has a unit processing time. Ambühl et al. (2011) prove that $1|prec|\sum w_i C_i$ remains strongly NP-hard if the precedence constraints form an interval order. Some polynomially solvable cases have been studied by Horn (1972) and Sidney (1975), who present an $O(n \log n)$ algorithm when G is a rooted tree, and by Adolphson (1977), who describes an $O(n \log n)$ algorithm when G is a series-parallel graph. Ambühl et al. (2011) exploit the relationship between the dimension theory of partial orders and $1|prec|\sum w_i C_i$ to obtain a polynomial-time 4/3-

approximation algorithm when G is a convex bipartite graph or a unit interval graph, and to obtain a 3/2-approximation for an arbitrary interval graph. These approximation results improve previous results by Woeginger (2003) and are the currently best-known approximation ratios.

For ease of exposition, when the precedence constraints are represented by a bipartite graph $G = (V_1 \cup V_2, A)$ (thus for each $(i, j) \in A$, $i \in V_1$ and $j \in V_2$), we write $1|V_1 \cup V_2| \sum w_i C_i$. Now consider the special case where the weights of the jobs in V_1 are zero, which is denoted by $1|V_1 \cup V_2, w(V_1) = 0| \sum w_i C_i$ (for any index set $A \subseteq \{1, \ldots, n\}$ we define $w(A) = \sum_{i \in A} w_i$). The variant of $1|V_1 \cup V_2, w(V_1) = 0| \sum w_i C_i$ in which any job in V_2 can be executed as soon as at least one of its predecessors in V_1 has been processed (*or*-type precedence constraints (Gillies and Liu, 1995; Möhring et al., 2004)), is denoted by $1|V_1 \cup V_2, w(V_1) = 0, or | \sum w_i C_i$.

Lemma 4.1. *Problem 1 is equivalent to* $1|V_1 \cup V_2, w(V_1) = 0| \sum w_i C_i$*, and Problem 2 is equivalent to* $1|V_1 \cup V_2, w(V_1) = 0, or | \sum w_i C_i$.

Proof. Consider an instance of Problem 1 with n boxes and m group activities. We construct an equivalent instance of $1|V_1 \cup V_2, w(V_1) = 0| \sum w_i C_i$ with $m + n$ jobs, where the first m jobs, called *group-activity jobs*, correspond with the m group activities and belong to V_1 whereas the last n jobs, called *box jobs*, correspond with the n boxes and all belong to V_2. The weight w_i of group-activity job i $(i = 1, \ldots, m)$ is 0 whereas the weight w_i of box job i $(i = m + 1, \ldots, m + n)$ is the probability π_{i-m} that the object is in the box $i - m$. Next, the processing time p_i of a group-activity job i $(i = 1, \ldots, m)$ is exactly the cost R_i of group activity i whereas the processing time p_i of box job i $(i = m + 1, \ldots, m + n)$ is equal to the cost t_{i-m} of searching box $i - m$. It can be readily verified that the total weighted completion time of the constructed instance and the total expected cost of finding the hidden item in the instance of Problem 1 are the same. Finally, the bipartite precedence graph $G = (V_1 \cup V_2, A)$ is such that there is an arc from group-activity job i $(i = 1, \ldots, m)$ to box job j $(j = m+1, \ldots, m+n)$ if and only if box $j - m$ belongs to S_i (the subset of

boxes associated with group activity i). This construction is done in polynomial time with respect to the size of the instance (in this case $n + m$). We can revert this construction to build an instance of Problem 1 from a given instance of $1|V_1 \cup V_2, w(V_1) = 0|\sum w_i C_i$, but then the job weights must be scaled to guarantee that the box probabilities π_i sum to one (as explained in the second paragraph of this section). For disjunctive group activities (Problem 2), the same construction is valid, but now a job in V_2 can be started as soon as at least one of its predecessors in V_1 has been executed (*or*-type precedence constraints).

□

4.4 Complexity of Problem 1

We first observe that the algorithms developed by Adolphson (1977) and Horn (1972) cannot solve Problem 1 because the corresponding bipartite graph is neither always a rooted tree nor always a series-parallel graph. In this section, we prove that $1|V_1 \cup V_2, w(V_1) = 0|\sum w_i C_i$ is strongly NP-hard, even if each job in V_1 has a unit processing time and each job in V_2 has a zero processing time and a unit weight. We use a reduction from $1|prec, p_i = 1|\sum w_i C_i$, which is known to be NP-hard in the strong sense (Lenstra and Rinnooy Kan, 1978). From Lemma 4.1, Problem 1 is then also strongly NP-hard (see Corollary 4.3 below).

Theorem 4.2. *The problem $1|V_1 \cup V_2, w(V_1) = 0|\sum w_i C_i$ is strongly NP-hard, even if each job in V_1 has a processing time of one and each job in V_2 has a processing time of zero and a weight of one.*

Proof. Clearly, the decision variant of $1|V_1 \cup V_2, w(V_1) = 0|\sum w_i C_i$ belongs to the class NP. Consider an arbitrary instance $\mathcal{I}$ of $1|prec, p_i = 1|\sum w_i C_i$ with job set $J = \{1, \ldots, n\}$, where each $i \in J$ has a processing time $p_i = 1$ and a non-negative integer weight w_i. The precedence constraints are described by a (directed acyclic) graph $G(J, A)$. We now construct

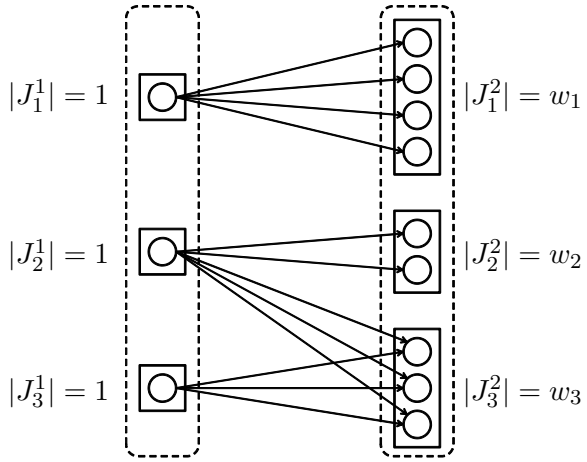


Figure 4.2: *Constructed instance $f(\mathcal{I})$ of $1|V_1 \cup V_2, w(V_1) = 0|\sum w_i C_i$ in proof of Theorem 4.2, where $\mathcal{I}$ is an instance of $1|prec, p_i = 1|\sum w_i C_i$ with $n = 3$, $w_1 = 4$, $w_2 = 2$, $w_3 = 3$, and $A = \{(2, 3)\}$*

an instance of $1|V_1 \cup V_2, w(V_1) = 0|\sum w_i C_i$, which we denote by $f(\mathcal{I})$, as follows. With each job $i \in J$ we associate two subsets J_i^1 and J_i^2, consisting of p_i and w_i copies of job i, respectively. Notice that $|J_i^1| = 1$ since the processing times p_i are unit processing times. We define $V_1 = \cup_{i \in J} J_i^1$ and $V_2 = \cup_{i \in J} J_i^2$. For each $i \in J$, the single job in J_i^1 has a weight of 0 and a processing time of 1, whereas for each job in J_i^2 we have a weight of 1 and a processing time of 0. Furthermore, there is an arc from the only job in J_i^1 to each job in J_j^2 when either $i = j$ or $(i, j) \in T(A)$, where $T(A)$ is the transitive closure of A. Figure 4.2 shows the constructed instance $f(\mathcal{I})$ for an example instance $\mathcal{I}$ of $1|prec, p_i = 1|\sum w_i C_i$ with $n = 3$, $w_1 = 4$, $w_2 = 2$, $w_3 = 3$, and $A = \{(2, 3)\}$. This completes the construction of the instance $f(\mathcal{I})$ of $1|V_1 \cup V_2, w(V_1) = 0|\sum w_i C_i$. Note that it is a pseudo-polynomial construction because it is polynomial in n and $\sum_{i \in J} w_i$. However, as $1|prec, p_i = 1|\sum w_i C_i$ is strongly NP-hard, it is sufficient to show that f is a pseudo-polynomial transformation (see (Garey and Johnson, 1979, p. 101)).

Note that any permutation of elements in $V_1 \cup V_2$ that satisfies the prece-

dence constraints is a feasible solution to $f(\mathcal{I})$. We now describe a subclass of feasible solutions to $f(\mathcal{I})$ that will be used to show the equivalence between $\mathcal{I}$ and $f(\mathcal{I})$: we only consider the solutions to $f(\mathcal{I})$ in which for each $i \in J$, the jobs in J_i^2 are scheduled consecutively and the single job in J_i^1 immediately precedes the block of jobs J_i^2. We call such a solution a *consecutive-index* solution. For the moment, let us assume that the following claim holds; its proof is established below.

Claim 1. *Any optimal solution to $f(\mathcal{I})$ is a consecutive-index solution.*

We now argue that any solution $(i_1, \ldots, i_n)$ to $\mathcal{I}$ can be transformed into a consecutive-index solution $(J_{i_1}^1, J_{i_1}^2, \ldots, J_{i_n}^1, J_{i_n}^2)$ to $f(\mathcal{I})$ with the same objective value, and vice versa. It can be verified that both schedules have a total weighted completion time equal to $\sum_{k=1}^{n} k w_{i_k}$ (recall that we have unit processing times in $\mathcal{I}$). Claim 1 together with this result will imply that $1|V_1 \cup V_2, w(V_1) = 0|\sum w_i C_i$ is at least as hard as $1|prec, p_i = 1|\sum w_i C_i$. Because the latter is strongly NP-hard, we conclude that Theorem 4.2 holds. □

Proof of Claim 1. We prove Claim 1 by induction on the the number of jobs $|J| = n$. When $n = 1$, all solutions to $f(\mathcal{I})$ are consecutive-index solutions, and take the form (J_1^1, J_1^2). Now assume that the claim holds whenever $n = r - 1$ with $r > 1$. In the remainder of this paragraph we will show that for any optimal solution α to $f(\mathcal{I})$ there is a job $i_1 \in J$ without predecessors in G such that the jobs in $J_{i_1}^1$ and $J_{i_1}^2$ are the first jobs in α, in that order. First note that it is a dominant decision to schedule a job of V_2 as soon as possible. Indeed, if we move a job of V_2 earlier in the schedule, we can only decrease its completion time, and since it has zero processing time, this will not increase the completion time of any other job in the schedule. As a result, jobs in any subset J_i^2 are scheduled consecutively in α. Let $J_{i_1}^2$ be the first block of jobs of V_2 in α. By construction, the job in $J_{i_1}^1$ and the jobs in any J_j^1 for which $(j, i_1) \in T(A)$, are scheduled before $J_{i_1}^2$. It can be seen that α cannot be optimal if i_1 has predecessors in G. Indeed, if i_1 has predecessors in G then there is a predecessor job j

without predecessors (G is acyclic). The schedule α can then be improved by first scheduling J_j^1 and J_j^2.

We conclude that $\alpha = (J_{i_1}^1, J_{i_1}^2, \alpha')$ where α' is an optimal solution to $f(\mathcal{I}')$ with $\mathcal{I}'$ the instance of $1|prec, p_i = 1|\sum w_i C_i$ obtained from $\mathcal{I}$ by removing i_1 from J together with all outgoing arcs. The instance $\mathcal{I}'$ has only $r-1$ jobs, and the associated precedence graph is again acyclic. By induction, α' is a consecutive index solution to $f(\mathcal{I}')$, thus we can write $\alpha' = (J_{i_2}^1, J_{i_2}^2, \ldots, J_{i_r}^1, J_{i_r}^2)$ with $\{i_2, \ldots, i_r\} = J \setminus \{i_1\}$. We conclude that $\alpha = (J_{i_1}^1, J_{i_1}^2, J_{i_2}^1, J_{i_2}^2, \ldots, J_{i_r}^1, J_{i_r}^2)$ is a consecutive-index solution to $f(\mathcal{I})$. □

Woeginger (2003) shows that the special case of $1|V_1 \cup V_2, w(V_1) = 0|\sum w_i C_i$ described in Theorem 4.2 is as hard to approximate as the general case $1|prec|\sum_i w_i C_i$. It should be noted that from this result, Theorem 4.2 can also be inferred. From Lemma 4.1 the following result ensues for Problem 1, and is valid even if all group activities have a unit cost, and all boxes are identical with a zero inspection cost.

Corollary 4.3. *Problem 1 is strongly NP-hard, even if $R_\ell = 1$ for all group activities ℓ, and for all the n boxes i we have $\pi_i = 1/n$ and $t_i = 0$.*

Corollary 4.3 implies that, unless P = NP, there is no (concise) LP-model for solving Problem 1. In particular, the integrality constraints in the integer-programming model proposed in Wagner and Davis (2001) cannot be relaxed, which reinforces the counterexample presented in Section 4.2.

An interesting special case of Problem 1 is the setting where a linear ordering of the boxes exists such that for each group activity ℓ, the associated boxes in S_ℓ are consecutive in this ordering. This geometrical property may be valid in several practical applications such as the presence of access covers over a set of consecutive machine components. The corresponding special case of $1|V_1 \cup V_2, w(V_1) = 0|\sum w_i C_i$ that is equivalent to this particular setting of Problem 1 is such that the bipartite precedence graph is *convex*. That is, the jobs in V_2 can be ordered such that for every job in V_1, the

successor jobs in V_2 are consecutive in that ordering. The polynomial-time 4/3-approximation algorithm that was presented in Ambühl et al. (2011) can be applied to approximate this special case of Problem 1. The complexity status of $1|prec|\sum_i w_i C_i$ with precedence constraints that form a convex bipartite order is still an open problem (Ambühl et al., 2011). Note that in this open problem, there is no restriction on the weights, whereas we demand zero weights for the jobs in V_1. Therefore, an NP-hardness result for the special case of $1|V_1 \cup V_2, w(V_1) = 0|\sum w_i C_i$ with convex bipartite precedence constraints would settle the open problem in Ambühl et al. (2011), whereas a positive result for the former problem would not necessarily apply for the latter.

4.5 Complexity of Problem 2

We now consider the disjunctive case (Problem 2), which is equivalent to $1|V_1 \cup V_2, w(V_1) = 0, or\,|\sum w_i C_i$, in which any job in V_2 can be executed as soon as at least one of its predecessors in V_1 has been processed (Lemma 4.1). The result of Corollary 4.3 does not apply to Problem 2 because it assumes that a box can be executed only when all its associated group activities have been processed. We show that $1|V_1 \cup V_2, w(V_1) = 0, or\,|\sum w_i C_i$ is strongly NP-hard using a reduction from the variant of 3-Dimensional Matching (3DM) defined as follows Garey and Johnson (1979):

3-Dimensional Matching (3DM):
Instance: A set $M \subseteq A \times B \times C$, where A, B, and C are disjoint sets having the same number q of elements and such that each element of A, B and C is the coordinate of at least one triple in M.
Question: Does M contain a matching, that is, a subset $M' \subseteq M$ such that $|M'| = q$ and no two elements of M' contain an identical coordinate?

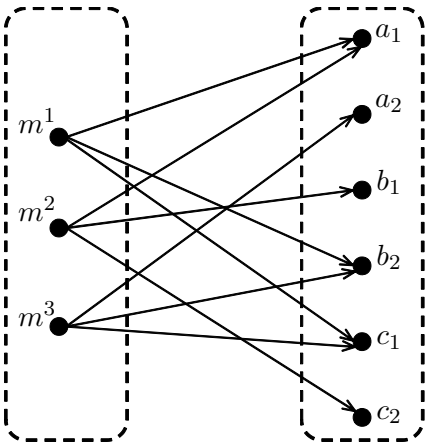


Figure 4.3: *Constructed instance of* $1|V_1 \cup V_2, w(V_1) = 0|\sum w_i C_i$ *in proof of Theorem 4.4 for* $q = 2$ *and* $M = \{m^1, m^2, m^3\}$*, with* $m^1 = (a_1, b_2, c_1)$*,* $m^2 = (a_1, b_1, c_2)$ *and* $m^3 = (a_2, b_2, c_1)$

Theorem 4.4. $1|V_1 \cup V_2, w(V_1) = 0, or\,|\sum w_i C_i$ *is strongly NP-hard, even when all jobs in* V_1 *have unit processing times and exactly three successor jobs in* V_2*, and all jobs in* V_2 *have zero processing times and unit weights.*

Proof. Clearly, the decision variant of $1|V_1 \cup V_2, w(V_1) = 0, or\,|\sum w_i C_i$ belongs to the class NP. Consider an arbitrary instance of 3DM described by the three distinct sets $A = \{a_1, \ldots, a_q\}$, $B = \{b_1, \ldots, b_q\}$, $C = \{c_1, \ldots, c_q\}$, and $M = \{m^1, \ldots, m^{|M|}\}$, where $m^i = (a^i, b^i, c^i) \in A \times B \times C$. We build an instance of $1|V_1 \cup V_2, w(V_1) = 0, or\,|\sum w_i C_i$ with $V_1 = \{m^1, \ldots, m^{|M|}\}$ containing $|M|$ elements, each corresponding with one $m^i \in M$. The set $V_2 = \{a_1, \ldots, a_q, b_1, \ldots, b_q, c_1, \ldots, c_q\}$ contains $3q$ jobs. Each job $m^i \in V_1$ is the predecessor of jobs $a^i, b^i, c^i \in V_2$. Furthermore, each job $m^i \in V_1$ has a weight of $w_{m^i} = 0$ and a processing time of $p_{m^i} = 1$, whereas each job $e \in V_2$ has a weight of $w_e = 1$ and a processing time of $p_e = 0$. This completes the description of our instance of $1|V_1 \cup V_2, w(V_1) = 0, or\,|\sum w_i C_i$. This construction can be set up in polynomial time. Figure 4.3 illustrates this construction for $q = 2$, $M = \{m^1, m^2, m^3\}$ with $m^1 = (a_1, b_2, c_1)$, $m^2 = (a_1, b_1, c_2)$ and $m^3 = (a_2, b_2, c_1)$. It can be easily verified that this is

a yes-instance.

We now argue that this instance of $1|V_1 \cup V_2, w(V_1) = 0, or \,|\sum w_i C_i$ has a solution with an objective value less than or equal to $\frac{3}{2}q(q+1)$ if and only if the instance of 3DM is a yes-instance. On the one hand, suppose that we have a yes-instance of 3DM; in other words, M contains a matching $M' = \left\{m^1, \ldots, m^q\right\}$ (up to a permutation of indices). We consider the following solution to $1|V_1 \cup V_2, w(V_1) = 0, or \,|\sum w_i C_i$: we first schedule the job m^1 followed by the three successor jobs a^1, b^1, c^1; we proceed with job m^2 and the three successor jobs a^2, b^2, c^2. This schedule continues up to job m^q and the three successor jobs a^q, b^q, c^q. The remaining jobs $m^{q+1}, \ldots, m^{|M|}$ are scheduled afterwards. This solution yields an objective value of $1 \times 3 + 2 \times 3 + \ldots + q \times 3 = \frac{3}{2}q(q+1)$.

On the other hand, suppose that $T = (t_1, \ldots, t_{3q+|M|})$ is a solution to $1|V_1 \cup V_2, w(V_1) = 0, or \,|\sum w_i C_i$ with an objective value less than or equal to $\frac{3}{2}q(q+1)$. Because of the precedence constraints, at least q jobs in V_1 must be scheduled before the last job in V_2 is scheduled. If $q+1$ jobs in V_1 are scheduled before the last job in V_2 then at least one job in V_2 is scheduled after the $q+1^{\text{th}}$ job in V_1. For that job, the weighted completion time is $1 \times (q+1)$. For the remaining $3q-1$ jobs scheduled before the $q+1^{\text{th}}$ job in V_1, the sum of weighted completion times is at least $1 \times 3 + 2 \times 3 + \ldots + (q-1) \times 3 + q \times 2$. Indeed, since we can sequence at most three consecutive jobs in V_2 (because every job in V_1 is the predecessor job of exactly three jobs in V_2), the best possible sequence holds exactly three consecutive jobs in V_2 immediately scheduled after the first $q-1$ jobs of V_1 such that the remaining two jobs in V_2 can be scheduled immediately after the q^{th} job in V_1. Summing up everything, we have $1 \times 3 + 2 \times 3 + \ldots + q \times 2 + q + 1 = 1 + \frac{3}{2}q(q+1) > \frac{3}{2}q(q+1)$. Therefore, any solution to $1|V_1 \cup V_2, w(V_1) = 0, or \,|\sum w_i C_i$ with an objective value less than or equal to $\frac{3}{2}q(q+1)$ executes exactly q jobs in V_1 before the last job in V_2. Since there are $3q$ jobs in V_2 and each job in V_1 is the predecessor of exactly three jobs in V_2, we infer that the schedule $T = (t_1, \ldots, t_{3q+|M|})$ is such that each of the q first scheduled jobs in V_1 is immediately followed

by three jobs in V_2. We consider the subset $M' \subseteq M$ built as follows: a triple m_i belongs to M' if and only if the corresponding job in V_1 is scheduled among the first q such jobs. It is not difficult to see that M' is a matching, which implies that we have a yes-instance of 3DM. This concludes the proof. □

From Lemma 4.1, we have the following complexity result for Problem 2.

Corollary 4.5. *Problem 2 is strongly NP-hard, even when* $R_\ell = 1$ *and* $|S_\ell| = 3$ *for each group activity* ℓ*, and* $\pi_i = 1/n$ *and* $t_i = 0$ *for each box* i*.*

In other words, Corollary 4.5 applies to the case where all group activities have a unit cost and exactly three associated boxes, and all boxes are identical with a zero inspection cost.

4.6 Some easy subproblems

In this section, we identify special cases of Problem 1 and Problem 2 that can be solved in polynomial time.

4.6.1 $S_\ell \cap S_{\ell'} = \emptyset$ for any two group activities ℓ and ℓ'

In this case there is no difference between the conjunctive and the disjunctive case, thus Problem 1 and Problem 2 coincide. The precedence graph of the equivalent scheduling problem is a forest of depth two. Therefore, we can solve this special case with Horn's algorithm for a forest (see Horn (1972)). The time complexity is $O(n \log n)$.

4.6.2 $|S_\ell| = 1$ for each group activity ℓ

For the conjunctive case the group activities can now be eliminated by adding to each box cost t_i the group activity costs R_ℓ for which $i \in S_\ell$. Next we may again apply Smith's rule. Alternatively, since the precedence

graph of the equivalent scheduling problem is an upside-down forest (of depth two), we may also apply Horn's algorithm for upside-down forests (Horn, 1972). For the disjunctive case in each rooted upside-down tree, we keep only one edge (ℓ, i) with smallest cost R_ℓ, reducing the problem to the previous case (Section 4.6.1). The time complexity for the conjunctive as well as the disjunctive case is again $O(n \log n)$.

4.6.3 $|S_\ell| = 2$ for each group activity ℓ

We will show that Problem 2 can be solved in polynomial time under the assumptions of Corollary 4.5 when in addition each group activity has exactly two associated boxes (instead of three). This result is true even if the cost of each group activity is different from one (but the same) and the cost of searching a box is different from zero (but the same). We call a box that does not appear in any subset S_ℓ a *free* box. Furthermore, a solution to an instance of Problem 2 is called a *maximal* solution when it has the form $(\alpha_2, \alpha_1, \alpha_0)$, where each group activity ℓ in α_k is immediately followed by exactly k boxes from S_ℓ $(k = 0, 1, 2)$. We need the following lemma.

Lemma 4.6. *Any optimal solution to an instance of Problem 2 without free boxes, with $R_\ell = 1$ and $|S_\ell| = 2$ for each group activity ℓ, and with $\pi_i = 1/n$ and $t_i = 0$ for each box i, is a maximal solution.*

Proof. First observe that, since searching a box has a zero cost, it is a dominant decision to search a box as soon as it is available. Therefore we may assume that in any optimal schedule, all unsearched boxes in a set S_ℓ are immediately searched after group activity ℓ is performed. Since each group activity is associated with exactly two boxes, in any optimal solution each group activity is followed by either two, one or zero boxes. Unexecuted group activities ℓ for which all boxes in S_ℓ have already been searched, can be placed at the end of the schedule without increasing the cost. Since we assume that the instance contains no free boxes, it remains

to be shown that in an optimal solution we should always search the boxes that can be sequenced in pairs before the boxes that can only be searched one by one. Assume, by contradiction, that there is an optimal schedule for which this is not true. In this schedule we identify the first group activity ℓ that is followed by both boxes in S_ℓ but for which the preceding k boxes are all immediately preceded by only one group activity ($k \geq 1$). If we move ℓ and S_ℓ directly before the group activity that precedes the first of these k boxes, the expected cost of each of the two boxes in S_ℓ decreases with k, whereas the expected cost of each of the k boxes increases by one. This operation thus results in a net decrease of the total expected cost equal to k with $k \geq 1$, and therefore it cannot be optimal. □

Theorem 4.7. *Problem 2 is polynomially solvable when each group activity ℓ has equal cost $R_\ell := R$ and $|S_\ell| = 2$, and all boxes i are identical with $\pi_i = 1/n$ and $t_i := t$.*

Proof. Without loss of generality, we may assume that $R = 1$ and $t = 0$. Indeed, if z is the expected cost of any feasible solution when $R = 1$ and $t = 0$, then the expected cost of that same solution for any value of R and t can be written as $Rz + t(n + 1)/2$, thus the optimal sequence remains unchanged if $R = 1$ and $t = 0$. The former expression for the expected cost holds because the contribution of the boxes to the expected cost is independent of the sequence and is equal to $(1/n)(t + 2t + \ldots + nt) = t(n + 1)/2$. We may also assume that there are no free boxes since these can always be scheduled first at zero cost.

Let $\mathcal{I}$ be an instance of Problem 2 that meets all these requirements. In other words, for instance $\mathcal{I}$, each group activity has a unit cost and is associated with exactly two boxes. Furthermore, all boxes are identical with zero inspection cost, and none of the boxes are free boxes. For instance $\mathcal{I}$, we construct an undirected graph G in which each node corresponds with a box, and where there is an edge between two nodes i and j if and only if there is a group activity ℓ such that $S_\ell = \{i, j\}$. The graph G can be constructed in polynomial time. In the remainder of this proof, we will

show that an optimal solution to $\mathcal{I}$ can be constructed in polynomial time by finding a maximum-cardinality matching in the graph G.

In this paragraph, we show that with each maximal solution $\alpha = (\alpha_2, \alpha_1, \alpha_0)$ to $\mathcal{I}$, we can associate a maximal matching M in G, and vice versa. On the one hand, for a maximal solution $\alpha = (\alpha_2, \alpha_1, \alpha_0)$, an edge $\{i, j\}$ of G belongs to M if i and j are consecutive boxes in α_2. Edges of M cannot have a node in common because otherwise α_2 would contain a group activity that is immediately followed by only one box. Therefore, M is a matching. The matching M is maximal because α_1 and α_0 do not contain two consecutive boxes. On the other hand, let M be a maximal matching of G with cardinality b. By construction, there are b distinct group activities, say $\ell_1, \ldots, \ell_b$, such that $M = \{S_{\ell_1}, \ldots, S_{\ell_b}\}$. From this matching M, we can construct a feasible solution α to $\mathcal{I}$ that is maximal, as follows. First, we execute the group activities ℓ_i immediately followed by a search of the two boxes in S_{ℓ_i}, for each $i = 1, \ldots, b$. The order in which the group activities are performed or the two boxes in a subset S_ℓ are searched is of no importance because this will not change the expected cost of the solution. Since M is a maximal matching and we have assumed that the instance contains no free boxes, there can be no two boxes of the remaining $n - 2b$ boxes that are searched consecutively. Moreover, for each of those $n - 2b$ unsearched boxes, there exists a different group activity that is not yet scheduled. We may complete the solution α by scheduling each of those group activities, each time followed by a search of the corresponding box. Finally, the remaining group activities are scheduled, if any are left. The construction of α can be done in polynomial time.

Finally, we will prove that a maximal solution to $\mathcal{I}$ is optimal if and only if the associated maximal matching in G is a maximum-cardinality matching. Since the maximum-cardinality matching problem is known to be polynomially solvable (Edmonds, 1965), the theorem then follows from Lemma 4.6. The total expected cost of the constructed maximal solution is $\frac{1}{n}(\sum_{i=1}^{b} 2i + \sum_{i=1}^{n-2b}(b+i))$, which equals $\frac{1}{n}(b(b+1) + (n-2b)(n+1)/2)$. By eliminating constant terms and factors, we can see that minimizing

this function is equivalent to minimizing $b(b-n)$. The latter function is monotone decreasing in b when $b \leq n/2$. Since the number of edges b in a matching of an undirected graph with n nodes is bounded by $\lfloor n/2 \rfloor$, minimizing $b(b-n)$ is equivalent to maximizing b. □

4.7 Conclusion and outlook

In this chapter, we have studied the computational complexity of the discrete sequential search problem with group activities, a problem that was first introduced by Wagner and Davis (2001), who presented an integer programming model for this problem. From their experiments, they conjecture that the conjunctive case can be solved as a linear program. First we prove this conjecture to be untrue by means of a counterexample. Next, we show that the sequential search problem with group activities can be viewed as a single machine scheduling problem with total weighted completion time objective on a bipartite graph with zero weights for the jobs in the first set. This implies that any result that holds for the search problem will also hold for the scheduling problem, and vice versa. We prove that the conjunctive case of the search problem is NP-hard; this corresponds with 'and' precedence constraints in the equivalent scheduling problem. Unless P = NP, this result implies that there can be no other (compact) linear program to solve the conjunctive case, hereby generalizing our findings in Section 4.2. We also show that the disjunctive case is difficult. The disjunctive search problem corresponds with 'or' precedence constraints in the equivalent scheduling problem. Both complexity results are true even in the setting where all group activities are identical, and all boxes are identical. For the disjunctive case, the problem even remains difficult when we also demand that each group activity be associated with exactly three boxes. The disjunctive case becomes easy, however, when there are only two boxes associated with each group activity.

As future work, we would like to settle the conjunctive case when the number of boxes associated with each group activity is two and three,

respectively. If we can show that the latter problem is easy, this would show that, in terms of computational complexity, the disjunctive case is more difficult than the conjunctive case. We would also like to discover the complexity status of the special case of Problem 1 described at the end of Section 4.4, where the equivalent scheduling problem has a convex bipartite precedence graph. It is worth noting that the instances of Problem 1 for which Wagner and Davis (2001) tested their LP model all have this property, and so their experimental findings may suggest that a polynomial algorithm exists in this particular setting. It is interesting to see if the result of Corollary 4.5 remains valid when convexity is added to the problem structure. Finally, we plan to verify whether Theorem 4.7 can be extended to the case of arbitrary probabilities.

List of Figures

List of Tables

Bibliography

Adolphson, D., 1977. Single machine job sequencing with precedence constraints. SIAM Journal on Computing 6, 40–54.

Ahlswede, R., Wegener, I., 1987. Search problems. John Wiley & Sons, Chichester.

Ambühl, C., Mastrolilli, M., Mutsanas, N., Svensson, O., 2011. On the approximability of single-machine scheduling with precedence constraints. Mathematics of Operations Research 36 (4), 653–669.

Assaf, D., Sharlin-Bilitzky, A., 1994. Dynamic search for a moving target. Journal of Applied Probability 31, 438–457.

Assaf, D., Zamir, S., 1987. Continuous and discrete search for one of many objects. Operations Research Letters 6, 205–209.

Aytug, H., Lawley, M., McKay, K., Mohan, S., Uzsoy, R., 2005. Executing production schedules in the face of uncertainties: A review and some future directions. European Journal of Operational Research 165 (1), 86–110.

Baldwin, C., Clark, K., 2000. Design Rules: The Power of Modularity. The MIT Press, Cambridge MA, USA.

Ben-Dov, Y., 1981a. A branch and bound algorithm for minimizing the expected cost of testing coherent systems. European Journal of Operational Research 7 (3), 284–289.

Ben-Dov, Y., 1981b. Optimal testing procedures for special structures of coherent systems. Management Science 27 (12), 1410–1420.

Birnbaum, Z., 1969. On the importance of different components in a multicomponent system. In: Krishnaiah, P. (Ed.), Multivariate Analysis-II. Vol. 1913. Academic Press, New York, pp. 15–26.

Boros, E., Ünlüyurt, T., 1999. Diagnosing double regular systems. Annals of Mathematics and Artificial Intelligence 26 (1-4), 171–191.

Butterworth, R., 1972. Some reliability fault-testing models. Operations Research 20 (2), 335–343.

Chun, Y., 1994. Sequential decisions under uncertainty in the R&D project selection problem. IEEE Transactions on engineering management 41 (4), 404–413.

Coolen, K., Talla Nobibon, F., Leus, R., 2013. Complexity analysis of the discrete sequential search problem with group activities. Tech. Rep. KBI_1313, Department of Decision Sciences and Information Management, FBE, KU Leuven.

Coolen, K., Wei, W., Talla Nobibon, F., Leus, R., 2011. Project scheduling with modular project completion on a bottleneck resource. Tech. Rep. KBI_1124, Department of Decision Sciences and Information Management, FBE, KU Leuven.

Coolen, K., Wei, W., Talla Nobibon, F., Leus, R., 2014. Scheduling modular projects on a bottleneck resource. Journal of Scheduling 17 (1), 67–85.

Cormen, T., Leiserson, C., Rivest, R., 1990. Introduction to Algorithms. MIT Press, Cambridge MA and McGraw-Hill, New York.

Creemers, S., Leus, R., Lambrecht, M., 2010. Scheduling Markovian PERT networks to maximize the net present value. Operations Research Letters 38, 51–56.

Creemers, S., Reyck, B. D., Leus, R., 2013. Project planning with alternative technologies in uncertain environments. Tech. Rep. 1314, Department of Decision Sciences and Information Management, FBE, KU Leuven.

Davenport, A., Beck, J., 2000. A survey of techniques for scheduling with uncertainty, unpublished manuscript, available at website `http://www.eil.utoronto.ca/chris/chris.papers.html`.

De, P., Dunne, E., J.B., G., C.E., W., 1995. The discrete time/cost trade-off problem revisited. European Journal of Operational Research 81, 225–238.

De Reyck, B., Demeulemeester, E., 1998. Local search methods for the discrete time/resource trade-off problem in project networks. Naval Research Logistics Quarterly 45, 553–578.

De Reyck, B., Grushka-Cockayne, Y., Leus, R., 2007. A new challenge in project scheduling: The incorporation of activity failures. Review of Business and Economics LII (3), 411–434.

De Reyck, B., Leus, R., 2008. R&D-project scheduling when activities may fail. IIE Transactions 40 (4), 367–384.

Demeulemeester, E., Vanhoucke, M., Herroelen, W., 2003. A random generator for activity-on-the-node networks. Journal of Scheduling 6, 13–34.

Dobbie, J., 1975. Search for an avoiding target. SIAM Journal on Applied Mathematics 28 (1), 72–86.

Drexl, A., 1991. Scheduling of project networks by job assignment. Management Science 37 (12), 1590–1602.

Edmonds, J., 1965. Paths, trees, and flowers. Canadian Journal of Mathematics 17, 449–467.

Garey, M., Johnson, D., 1979. Computers and Intractability. A Guide to the Theory of NP-completeness. Freeman, San Francisco.

Gillies, D., Liu, J., 1995. Scheduling tasks with AND/OR precedence constraints. SIAM Journal on Computing 24, 797–810.

Gittins, J., 1989. Multi-Armed Bandit Allocation Indices. John Wiley & Sons, Chichester.

Gluss, B., 1959. An optimum policy for detecting a fault in a complex system. Operations Research 7, 468–477.

Graham, R., Lawler, E., Lenstra, J., Rinnooy Kan, A., 1979. Optimization and approximation in deterministic sequencing and scheduling: A survey. Annals of Discrete Mathematics 5, 287–326.

Herroelen, W., Leus, R., 2005. Project scheduling under uncertainty: Survey and research potentials. European Journal of Operational Research 165 (2), 289–306.

Horn, W., 1972. Single machine job sequencing with treelike precedence ordering and linear delay penalties. SIAM Journal of Applied Mathematics 23 (2), 189–202.

Huysmans, M., Coolen, K., Talla Nobibon, F., Leus, R., 2012. A fast greedy heuristic for scheduling modular projects. Tech. Rep. KBI_1227, Department of Decision Sciences and Information Management, FBE, KU Leuven.

Jain, V., Grossmann, I., 1999. Resource-constrained scheduling of tests in new product development. Industrial and Engineering Chemistry Research 38, 3013–3026.

Jędrzejowicz, P., 1983. Minimizing the average cost of testing coherent systems: Complexity and approximate algorithms. IEEE Transactions on Reliability R-32 (1), 67–70.

Johnson, D., Yannakakis, M., Papadimitriou, C., 1988. On generating all maximal independent sets. Information Processing Letters 27, 119–123.

Joyce, W., 1971. Organization of unsuccessful R&D programs. IEEE Transactions on Engineering Management 18 (2), 57–65.

Kavadias, S., Loch, C., 2003. Optimal project sequencing with recourse at a scarce resource. Production and Operations Management 12 (4), 433–444.

Kelly, F., 1982. A remark on search and sequencing problems. Mathematics of Operations Research 7 (1), 154–157.

Knuth, D., 1998. The Art of Computer Programming, Volume 3: Sorting and Searching, 2nd Edition. Addison-Wesley.

Kolisch, R., 1996a. Efficient priority rules for the resource-constrained project scheduling problem. Journal of Operations Management 14, 172–192.

Kolisch, R., 1996b. Serial and parallel resource-constrained project scheduling methods revisited: Theory and computation. European Journal of Operational Research 90, 320–333.

Kulkarni, V., Adlakha, V., 1986. Markov and Markov-regenerative PERT networks. Operations Research 34, 769–781.

Lenstra, J., Rinnooy Kan, A., 1978. Complexity of scheduling under precedence constraints. Operations Research 26 (1), 22–35.

Malewicz, G., 2005. Parallel scheduling of complex dags under uncertainty. In: Proceedings of the 17th Annual ACM Symposium on Parallel Algorithms. pp. 66–75.

Mitten, L., 1960. An analytic solution to the least cost testing sequence problem. The Journal of Industrial Engineering January-Februari, 17.

Möhring, R., 2000. Scheduling under uncertainty: Optimizing against a randomizing adversary. In: Jansen, K., Khuller, S. (Eds.), Approximation Algorithms for Combinatorial Optimization, Proceedings of the

Third International Workshop APPROX 2000, Saarbrücken. Vol. 1913 of Lecture Notes in Computer Science. Springer-Verlag, pp. 15–26.

Möhring, R., Skutella, M., Stork, F., 2004. Scheduling with AND/OR precedence constraints. SIAM Journal on Computing 33 (2), 393–415.

Monma, C., Sidney, J., 1979. Sequencing with series-parallel precedence constraints. Mathematics of Operations Research 4 (3), 215–224.

Oliveira, S., Stewart, D., 2006. Writing Scientific Software: A Guide to Good Style. Cambridge University Press.

Owen, G., McCormick, G., 2008. Finding a moving fugitive. a game theoretic representation of search. Computers & Operations Research 35 (6), 1944–1962.

Puterman, M., 1994. Markov Decision Processes: Discrete Stochastic Dynamic Programming. John Wiley and Sons.

Radermacher, F., 1981. Cost-dependent essential systems of ES-strategies for stochastic scheduling problems. Methods of Operations Research 42, 17–31.

Ranjbar, M., Davari, M., 2013. An exact method for scheduling of the alternative technologies in R&D projects. Computers and Operations Research 40, 395–405.

Runarsson, T., Yao, X., 2000. Stochastic ranking for constrained evolutionary optimization. IEEE Transactions on Evolutionary Computation 4 (3), 284–294.

Sabuncuoglu, I., Goren, S., 2009. Hedging production schedules against uncertainty in manufacturing environment with a review of robustness and stability research. International Journal of Computer Integrated Manufacturing 22 (2), 138–157.

Schmidt, C., Grossmann, I., 1996. Optimization models for the scheduling of testing tasks in new product development. Industrial and Engineering Chemistry Research 35, 3498–3510.

Sidney, J., 1975. Decomposition algorithms for single machine scheduling with precedence relations and deferral costs. Operations Research 23 (2), 283–298.

Smith, W., 1956. Various optimizers for single-stage production. Naval Research Logistics Quarterly 3, 59–66.

Song, N.-O., Teneketzis, D., 2004. Discrete search with multiple sensors. Mathematical Methods of Operations Research 60, 1–13.

Standish, T., 1995. Data Structures, Algorithms & Software Principles in C. Addison-Wesley.

Stone, L., 1989. Whats happened in search theory since the 1975 lanchester prize. Operations Research 36, 501–506.

Stork, F., 2001. Stochastic resource-constrained project scheduling. Ph.D. thesis, TU Berlin, Germany.

Talbot, F., 1982. Resource-constrained project scheduling problem with time-resource trade-offs: the nonpreemptive case. Management Science 28, 1197–1210.

Talla Nobibon, F., Coolen, K., Leus, R., 2013. A note on 'discrete sequential search with group activities'. Decision Sciences 44 (2), 395–401.

Ünlüyurt, T., 2004. Sequential testing of complex systems: A review. Discrete Applied Mathematics 142, 189–205.

Valdes, J., Tarjan, R., Lawler, E., 1982. The recognition of series parallel digraphs. SIAM Journal on Computing 11 (2), 298–313.

Vieira, G., Herrmann, J., Lin, E., 2003. Rescheduling manufacturing systems: A framework of strategies, policies, and methods. Journal of Scheduling 6 (1), 39–62.

Wagner, B., Davis, D., 2001. Discrete sequential search with group activities. Decision Sciences 32 (4), 557–573.

Wang, T., 2007. Integer hash function. Tech. rep., HP Enterprise Java Lab, available at website `http://www.concentric.net/ ttwang/tech/inthash.htm`.

Weber, R., 1986. Optimal search for a randomly moving object. Journal of Applied Probability 23, 708–717.

Woeginger, G., 2003. On the approximability of average completion time scheduling under precedence constraints. Discrete Applied Mathematics 131, 237–252.

Doctoral dissertations from the Faculty of Business and Economics

A list of doctoral dissertations from the Faculty of Business and Economics can be found at the following website:
`http://www.kuleuven.ac.be/doctoraatsverdediging/archief.htm`.